APPENDIX 2

Table of Contents

1.0 Introduction

In the *Script Development* manual, you learned how to create sophisticated scripts for emulating user activity on your system under test. This manual guides you through the execution of multi-user tests with EMPOWER, EMPOWER/X, or EMPOWER/CS.

For multi-user testing, you may execute several scripts simultaneously to apply a workload to the system under test. Individual scripts may be duplicated to represent multiple users performing similar activities, and different scripts may be mixed to emulate a complex set of user and application activity.

1.1 Multi-User Testing Tools

Multi-user testing is conducted with the following tools: Mix, Extract, Report, Draw, Monitor, and Global Variables (GV).

Mix emulates an actual benchmark environment by executing multiple scripts. It uses a script table and, optionally, a command file to specify the number of emulated users, the name of each executed script, each user's communication line, the delay time between the start of each script, and the termination condition of the emulation. Mix executes compiled scripts and each executing script produces a log file used to prepare resulting reports.

Generating performance reports is accomplished in two steps. The Extract tool parses the log file of each script and records response time information in a set of flat ASCII files that are used as input for the Report tool. (You also may choose to generate reports from your own statistical or graphics programs.)

Report reads the ASCII files generated by Extract and produces statistical reports describing response time and system throughput. The reports can be generated to conform to reporting specifications often defined in Government Services Administration benchmarking requirements (see GSA FPR 1-4.11).

After you have generated performance reports for your multi-user emulation, Draw will accept one or more reports to produce bar charts that depict relationships among performance results. These relationships can summarize a single multi-user test or a series of multi-user tests in which each test contains a different user level or system configuration.

The Monitor modules display critical information about emulated users during multi-user benchmarks. A single ASCII terminal, X terminal, or workstation is used to monitor response times, control script execution, track and correct script errors and timeouts, and present benchmark progress. Information is provided in logically grouped views that may be sorted on any field in the view. With the Monitor tool, you can identify and debug problem scripts, verify system load, and take a snapshot of user activity during a system bottleneck. Commands are provided to suspend, resume, and kill scripts during a multi-user test.

The Global Variables tool, EMPOWER/GV provides advanced control over multi-user emulations by creating and controlling global variables that are shared among executing scripts. Variables can be defined to direct scripts to terminate gracefully when they run an indefinite process. They can be used as a counter for scripts that must perform a fixed amount of work before terminating. Variables also may be used to synchronize the execution of multiple scripts. The elements of EMPOWER/GV include commands and functions which control access to a variable; read, update, or test values of variables; or control a shared memory segment.

1.2 User's Guide Organization

General use information and specific technical reference material for operating the PERFORMIX load testing products are provided in several manuals. Examples have been provided for clarity.

This manual, *Multi-User Testing*, is divided into the following sections:

Section 1:	Provides an introduction to multi-user testing
Section 2:	Describes the Mix tool which combines scripts to develop complete emulations of systems or applications to be tested
Section 3:	Describes the Extract tool which collects performance data from log files of executed scripts
Section 4:	Describes the Report tool which produces statistical reports of results of multi-user performance tests
Section 5:	Describes the Draw tool which graphically displays results of multi-user performance tests
Section 6:	Describes the Monitor tool which displays critical information about scripts during execution. It also can control script execution

Note: This manual is used for the PERFORMIX Inc. products EMPOWER, EMPOWER/X, and EMPOWER/CS. The EMPOWER, EMPOWER/X, and EMPOWER/CS tools for script development are all maintained separately. However, the tools for multi-user testing, as described in this manual, are common among the EMPOWER products. Some commands and situations may apply only to one of the products. These instances are specifically noted within the text.

You may notice different version numbers for any one of the multi-user tools. If you have purchased two or more EMPOWER products, the version number for the multi-user testing tools will correspond to one of the EMPOWER products purchased.

1.3 User's Guide Conventions

The conventions followed in this User's Guide are listed below:

Regular Font	Used for all regular body text
`Mono-spaced Font`	Used for all command, function, and file names; for all examples; and, generally, for any computer-generated text
`Bold Mono-spaced Font`	In examples, represents entries made by the EMPOWER, EMPOWER/X, or EMPOWER/CS user
`[-p port]`	In command syntaxes, text within brackets represents optional command parameters
`gv_stat[name | -s]`	Vertical lines (|) separate command parameters
`...`	Within scripts, the ellipsis marks indicate some script content was left out for brevity
`Beginfunction()`	Parentheses are included with script functions mentioned in regular body text. For most functions, one or more parameters will be listed in the parentheses
`Endscenario()`	EMPOWER/CS script functions use initial capitalization
`extract`	EMPOWER/CS command names use all lower case letters
Mix	When an EMPOWER, EMPOWER/X, or EMPOWER/CS tool is mentioned within regular body text, it is shown in regular font with initial capitalization

The term "SUT" refers to your system under test.

2.0 Mix

Complete, thorough emulations must stress a system under test (SUT) to verify that it can handle predicted workloads. Therefore, EMPOWER, EMPOWER/X, and EMPOWER/CS must be able to emulate various users operating multiple terminals connected to the SUT (or for EMPOWER/CS, multiple PCs connected to the server).

A goal for realistic load testing is to replace the environment of the SUT. This goal is accomplished with the Mix tool which combines multiple scripts to simulate an actual load on the system. Individual scripts may be executed in parallel by multiple emulated users, and different scripts may be mixed to represent complex user and application activity. To simulate the entire system environment, Mix simultaneously controls the complete mix of scripts.

2.1 The Mix Table

Specifications for a multi-user emulation are provided in a mix table which identifies users and specifies the script each user will emulate. Each line in the table directs the execution of a single script, and the size of the mix table is unlimited. In the following example mix table, we will emulate six users working simultaneously. An example for each EMPOWER product is included.

```
user1, query telnet:sut log1
user2, report tty8 log2
user3, xquery xquery log3
user4, xquery -d serverhost:0 xquery log4
user5, csquery log5
user6, csupdate log6
```

The first item you enter in a mix table is the script ID which represents the emulated user name. The script ID is used by the Mix and Monitor tools to identify each script individually. Following the user name and comma in each line is the script execution command which includes the executable script file name and any valid script execution options. These options are fully defined in the Compiled Script Execution sections of the EMPOWER *Script Development*, *EMPOWER/X Script Development*, and EMPOWER/CS *Script Development* manuals.

While executing scripts, Mix will create log files for each script which are named by adding a `.l` extension to the script ID. The default logs files for the following example scripts will be `user1.l`, `user2.l`, `user3.l`, and `user4.l`:

```
user1, script1 telnet:sut
user2, script1 telnet:sut
user3, script2 telnet:sut
user4, script2 telnet:sut
```

If `E_PORT` equals `telnet:sut`, you do not need to specify `telnet:sut`. The above Mix table could look like the following:

```
user1, script1
user2, script1
user3, script2
user4, script2
```

If you want to name the script log files different from the script id, you must type the names after the specified port.

Example:

```
user1, script1 telnet:sut log1
user2, script1 telnet:sut log2
user3, script2 telnet:sut log3
user4, script2 telnet:sut log4
```

If you wish to pass your own arguments to the script, you are required to specify the port and log for each script.

Example:

```
user1, script1 telnet:sut log1 user001 pass001
user2, script1 telnet:sut log2 user002 pass002
user3, script2 telnet:sut log3 user003 pass003
user4, script2 telnet:sut log4 user004 pass004
```

You may want a single emulated user to execute several scripts consecutively, which is achieved by replacing the user name for subsequent scripts with a "+", as shown in the following example. You will need to specify different log names or subsequent scripts will overwrite the default log files of the previous script execution.

```
u1, query telnet:sut log1a
+   report telnet:sut log1b
+   modify telnet:sut log1c
```

In the above example, one ASCII terminal user executes the script called `query`, then the script called `report`, then the script called `modify`. The results are saved in the Mix log files `log1a`, `log1b`, and `log1c`, respectively.

A similar process applies to an X terminal user in the following example:

```
u2, report report log2a
+   query  query  log2b
+   modify modify log2c
```

For a PC user:

```
u3, report log3a
+   query log3b
+   modify log3c
```

To simulate the time a user might spend between different operations (different scripts), you may add a "sleep" period before executing another script.

This activity is performed by adding a `sleep` parameter after the + in the mix table entry:

```
u1, query telnet:sut log1a
+  sleep 5 report telnet:sut log1b
+  sleep 5 modify telnet:sut log1c
```

After you have created your Mix table, you must save it as a file to be used later in your Mix emulation. The Mix table file name is specified in the `use` command of Mix. Refer to section 2.3.23 for a description of the `use` command.

2.2 Mix Syntax

Once the mix table has been created, the `mix` command is used to execute scripts. The syntax of the `mix` command is given in the following usage message. You may access the usage screen by entering the mix command with a hyphen parameter:

```
$ mix -

EMPOWER V3.1.8, Serial#R00000-000, Copyright PERFORMIX, Inc. 1988-95
Usage:
        mix [-f mixfile [-d]] [-l log] [-I]|[[-D] [-p port]]
Options:
        -f mixfile      Runs MIX in batch mode reading commands from mixfile
        -d         Prevents display of MIX session on screen
        -l log     Causes the MIX trace to be stored in log
        -p port    Specifies the port Daemon should listen on
Note:
        MIX runs in interactive mode by default.
        In interactive mode, type ? to obtain more help.
        Use the -f and -d options when running MIX in the background.
        By default, the MIX trace is stored in mix.log.
        Default MIX Daemon port is 7273.
Examples:
        mix
        mix -f run3user
        mix -f run6user -d &
        mix -l 3user.log
        mix -p 3333
```

0056754.0260

2.2.1 Batch Mode

When Mix runs in interactive mode, you can control execution of the multi-user test by entering Mix commands at the `mix>` prompt on the UNIX script driver machine. However, a large test may be time consuming. Therefore, you can execute a multi-user test by running Mix in batch mode. In batch mode, Mix will read commands from a command file rather than from your terminal.

Create the command file you wish to use in your emulation and then specify the command file name in the `-f` option's parameter (`mix -f cmdfile`). Mix will terminate when it reaches the end of the command file or when it encounters a `quit`, `q`, or `exit` command.

Typically, one command file is generated for each configuration to be tested. For example, if you are testing your SUT with one, eight, and thirty-two users, you might create three separate command files called `1user.cmd`, `8user.cmd`, and `32user.cmd` that would look like the following:

```
1user.cmd:               8user.cmd:               32user.cmd:

use 100 user.tab         use 8user.tab            use 100user.tab
start user001            start all                start user[001-032]
wait                     wait                     wait
quit                     quit                     quit
```

Note: Any lines in a batch file that start with "# " are assumed to be comments and are ignored during script execution.

2.2.2 Turning Off the Mix Display

When you execute Mix in batch mode, the output of commands executed from the command file is displayed at the UNIX script driver. If you do not want Mix to display this output, for example if Mix is running in the background, enter the `-d` option of the `mix` command.

Example:

```
$ mix -d
```

2.2.3 The Mix Log

When Mix executes a mix table, it generates a log of its activities. The Mix log file contains all executed Mix commands, the time that each command was executed, and a copy of the Mix output. If the log file already exists when Mix is run, it will be overwritten.

By default, this log is called `mix.log` and is saved in the current directory. If you wish to specify a different name, use the `mix` command's `-l` option with a parameter of the desired log file name. Typical names are `1user.log`, `8user.log`, and `32user.log`.

An example Mix log file is shown below:

```
15:38:27.53 mix> use tab
15:38:30.47 mix> set tstart 2
15:38:34.75 mix> start all
15:38:38.06 [user01] started
15:38:40.17 [user02] started
15:38:42.17 [user03] started
15:38:44.17 [user04] started
15:38:44.23 mix> wait
16:35:26.36 user01 terminated (3/4) running
16:35:26.36 user02 terminated (2/4) running
16:35:28.44 user03 terminated (1/4) running
16:35:28.44 user04 terminated (0/4) running
16:35:28.45 no more scripts running
16:35:28.45 mix> quit
```

2.2.4 Specifying Ports for Mix daemons

Relevant only to distributed emulations with Mix, the `-p` option of `mix` specifies the port that Mix daemons should listen on. The default Mix daemon port is 7273. Refer to Section 2.5.2 for a more detailed explanation of distributed emulations with Mix.

2.3 Mix Commands

The Mix tool allows you to control script execution in the mix table by providing several Mix commands. Mix is a command interpreter, so it displays its own prompt on the UNIX script driver when executed interactively. The Mix commands may be entered at this prompt or may be contained in command files for batch execution.

2.3.1 ?

The ? command displays the Mix command help screen:

```
$ mix
EMPOWER V3.1.8, Serial#R00000-000, Copyright PERFORMIX, Inc. 1988-95

mix> ?

command    syntax                description
-------    ------                -----------
at         at n cmd              run MIX command at n seconds
close      close                 forget about entries in last script table used
connect    connect host ...      connect to MIX daemon on host(s)
exec       exec commandfile      execute file of MIX commands
get        get host file ...     copy file(s) from host
kill       kill all | id ...     terminate execution of script(s)
pause      pause n               sleep for n seconds
put        put host file ...     copy file(s) to host
quit       quit                  exit MIX
exit       quit                  exit MIX
rcd        rcd host dir          change directory on host
rcmd       rcmd host cmd         run cmd on host
resume     resume all | id ...   continue execution of suspended scripts
set        set [variable value]  initialize a MIX variable
signal     signal all | id ...   send signal to script(s)
start      start all | id ...    begin execution of script(s)
stat       stat [-l] [id]        display script status
table      table [-a] name fmt   make a table using format
time       time                  display current time of day
trap       trap command          run command if !cmd exits with error
use        use table             read script table
wait       wait                  pause until all scripts complete
mix>
```

2.3.2 !

The ! command executes a shell command from within Mix on the UNIX driver machine. Upon completion of the shell command, control returns to Mix and Mix displays its prompt.

Examples:

```
mix> !date
Thu Oct  6 15:49:35 EDT 1995
mix> !ps
  PID TT STAT  TIME COMMAND
23917 p2 IW    0:00 -sh (sh)
25302 p2 S     0:00 mix
25311 p2 S     0:00 sh -c ps
25312 p2 R     0:00 ps
mix>
```

2.3.3 At

The `at` command is used to execute another Mix command at a specified time. The syntax of the `at` command is shown below:

```
at n command
```

The integer `n` represents the time to execute the command, and `command` is the Mix command to be executed. The time is specified in seconds from the beginning of the Mix session. The `at` command is useful for executing Mix in batch mode. For example, the following command will execute the Mix command `start all` one minute (60 seconds) after the Mix session has started:

```
mix> at 60 start all
```

If the `at` command executes after the specified time has expired, a warning will be displayed and the command will execute. The following example Mix session shows the use of the `at` command 30 seconds after the session began:

```
mix> at 5 time
warning: at time has passed
time:  12:11:05
```

2.3.4 Close

The close command ends the use of one mix table before proceeding to another mix table. A close command must be used before a second use command is entered. Refer to section 2.3.23 for a description of the use command.

Example:

```
mix> use table1
mix> start all
mix> wait
mix> close
mix> use table2
```

2.3.5 Connect

The connect command is used only when running Mix as a daemon on multiple UNIX driver machines. This command connects the specified remote Mix daemon(s) to the central Mix session. Refer to Sections 2.5.2 and 2.5.3 for a more detailed explanation of distributed emulations with Mix.

2.3.6 Exec

The exec command, generally used in interactive mode, directs Mix to retrieve and execute a series of commands that are stored in a separate file. This capability is helpful for executing a frequently used set of commands without constantly retyping them. Notice that the user controlling Mix types only two commands to start the test.

Example:

The s4 file:

```
use 4usertab
set tstart 2
set tresume 20
set logdir ./logs
!date
```

The Mix session:

```
mix> exec s4
mix> use 4usertab
mix> set tstart 2
mix> set tresume 20
mix> set logdir ./logs
mix> !date
Thu Oct  6 15:59:24 EDT 1995
mix> start all
```

2.3.7 Get

The `get` command is used when running Mix as a daemon on multiple UNIX script driver machines. This command copies a specified file(s) from a specified remote UNIX driver machine to the primary UNIX driver machine. Refer to Section 2.5.3 for a more detailed explanation.

2.3.8 Help

The `help` command displays the help menu. This command performs the same action as "?" .

2.3.9 Kill

The `kill` command terminates execution of one or more scripts. You can specify scripts to be killed by their script ids in the first field of the script file entry. The `kill` command is followed either by the script ids or by the key word `all`, which will terminate the execution of all scripts currently executing.

Example:

```
mix> use table1
mix> start all
mix> pause 600
mix> kill all
mix> wait
mix> quit
```

The `kill` command allows the specification of a range. For example, to kill the scripts `user01`, `user02`, ..., `user07`, the following command is used:

```
mix> kill user[01-07]
```

To specify multiple scripts that are not continuous, use the command specification shown in the following example:

```
mix> kill user[02,04,12]
```

Note: The specification of ranges in Mix does not work exactly like the specification of ranges in the UNIX shell. The following UNIX shell command would list `chap1` through `chap5` and `chap0`:

```
$ ls chap[1-50]
```

Range specification in the UNIX shell accepts only single characters.

2.3.10 Pause

The `pause` command causes Mix to delay for a period of seconds. `Pause` often is used in command files when an emulation must execute for a specified time before terminating. `Pause` can be used when a large number of scripts require time to log in and suspend themselves before all scripts can be resumed.

Example:

```
mix> use table9
mix> start all
mix> pause 20
mix> resume all
mix> wait
mix> quit
```

2.3.11 Put

The `put` command is used when running Mix as a daemon on multiple UNIX script driver machines. This command copies a specified file(s) from the primary UNIX driver machine to the specified remote machine. Refer to Section 2.5.3 for a more detailed explanation of distributed emulations with Mix.

2.3.12 Quit

The `quit` command causes Mix to terminate. If scripts are executing, Mix will ask for verification. If you decide to quit while scripts are executing, the scripts will continue until they terminate themselves. If Mix is executed in batch mode, the `quit` command will not ask for confirmation. (*Note:* You can also enter `q` or `exit` which function exactly as `quit`.)

Example:

```
mix> quit
$
```

2.3.13 Rcd

The `rcd` command is used when running Mix as a daemon on multiple UNIX script driver machines. This command causes the Mix daemon on the specified remote UNIX driver machine to change to a specified directory. Refer to Section 2.5.3 for a more detailed explanation of distributed emulations with Mix.

2.3.14 Rcmd

The `rcmd` command is used when running Mix as a daemon on multiple UNIX script driver machines. This command executes a specified command on a specified UNIX driver machine. Refer to Section 2.5.3 for a more detailed explanation.

2.3.15 Resume

The `resume` command causes Mix to resume suspended scripts. The `resume` command must be followed by a list of identifiers or the key word `all` which resumes all scripts.

A script execution is suspended when it executes the `Suspend()` function. The `Suspend()` function is useful for multi-user emulations that require all users to be logged into the SUT before transactions can be executed or response times can be measured.

When resuming more than one script, each script will resume after a delay interval which is determined by the value of the TRESUME variable. The default TRESUME value is five seconds. To change the value of TRESUME, use the Mix set command.

Example:

```
mix> use table1
mix> set TRESUME 10
mix> start user1 user2 user3
mix> pause 45
mix> resume all
mix> wait
mix> quit
```

The resume command allows the specification of a range. For example, to resume the scripts user01, user02, ..., user07, the following command is used:

```
mix> resume  user[01-07]
```

To specify multiple scripts that are not continuous, use the command specification shown in the following example:

```
mix> resume  user[02,04,12]
```

2.3.16 Set

The set command defines values for the Mix variables. It is followed by the name of the variable—TSTART, TRESUME, TSIGNAL, LOGDIR, TABLE, or CONTINUE—and the new value of the variable. If you enter the set command without any parameters, Mix will display the current value of all variables.

TSTART controls the delay between starting scripts. TRESUME controls the delay when resuming scripts. TSIGNAL controls the delay between signaling scripts. LOGDIR defines the directory where log files of scripts to be executed are stored.

TABLE indicates the mix table that will be used for the session. CONTINUE has two values (on or off) that indicate whether or not continuation scripts will be executed. (Continuation scripts begin with "+" in the Mix table.) All variables are recognized in lower case letters, for example, tstart and tresume.

The default values of TSTART and TRESUME are five seconds and TSIGNAL is zero seconds. If you prefer, you can specify these values as floating point numbers which allows you to start or resume scripts a half-second apart, one-and-a-quarter seconds apart, etc. The default value of LOGDIR is the current directory. TABLE has no default, and the default for CONTINUE is "on".

Examples:

```
mix> use 4 usertab
mix> set tstart 2
mix> set tresume 10
mix> set logdir ./logs
mix> set
TSTART   2.00
TRESUME  10.00
TSIGNAL  0.00
LOGDIR   ./logs
TABLE    4usertab
CONTINUE on
mix>
```

2.3.17 Signal

The Mix signal command sends a signal to a script to control execution. For example, a signal command can be used with the file input/output routines to read type rate values from another file. The script must include a Signal() function to specify an action to take when the script receives the signal. If the script does not include a Signal() function, the signal command will be ignored.

The syntax of the `signal` command is:

```
signal scriptid [...]
```

`scriptid` specifies a script where the signal should be sent. The parameter of `scriptid` may be "`all`", in which case the signal is sent to all scripts.

The following example shows a portion of a script that uses the `Signal()` function. When the Mix user enters a `signal` command, the type rate will be read from a file:

```
typechange()
{
int i;
      Fiorewind("/tmp/typerate");    /* go to beginning of file */
      Fioreadline("/tmp/typerate"); /* read line in */
      if(FIOLEN > 0) {               /* if no error */
            i=atoi(FIOBUFFER);       /* convert value to int */
            if(i > 0)
                  Typerate(i);       /* set the new type rate */
            else
                  Signal(IGNORE);    /* ignore further signals */
      }
}
Empower(argc, argv)
int argc;
char **argv;
{
Thinkuniform(1,2.5);
Timeout(300,EXIT);
Signal(typechange);      /* set the Signal handler */
Beginscenario("shell");
/* ...
rest of script
... */
Endscenario("shell");
```

When this script executes, the type rate initially is set to the default of zero characters per second. The first time the Mix `signal` command is entered at the UNIX script driver, the value in the file `/tmp/typerate` is read and, if the value is not zero, the type rate is set to the specified value. This process repeats every time

0969794.10260

the `signal` command is entered until the value read from the file is zero. At that point, all subsequent signals are ignored.

2.3.18 Start

The `start` command executes one or more scripts. To begin executing one or more scripts, enter the `start` command followed by the script ids from the mix table. To start all scripts in the mix table, enter the `start` command followed by the key word `all`.

When starting multiple scripts, each script will begin after a delay determined by the `TSTART` variable.

Examples:

```
mix> set TSTART 10
mix> start user1 user2 user3
[user1] started
[user2] started
[user3] started
```

The `start` command allows specification of a range. For example, to start the scripts `user01`, `user02`, ..., `user07`, enter the following command:

```
mix> start user[01-07]
```

To specify multiple scripts that are not continuous, such as user02, user04, and user12, use the following command format:

```
mix> start user[02,04,12]
```

2.3.19 Stat

The `stat` command displays the status of executing scripts. An asterisk (`*`) following a user name indicates that the script is executing. In the following example, three scripts were read in the mix table and only `user2's` script is running.

Example:

```
mix> stat
user1  user2*  user3
```

The `-l` option of the `stat` command displays more detailed status in four columns. The column headings are `script_id`, `process_id`, `sleep`, and `script`. `Script_id` is the user name defined in the mix table. `Process_id` is the process identifier assigned by the operating system for the script. The `process_id` will be -1 if a script is not executing. `Sleep` is the time a script will delay before it begins execution. `Script` is the script to be executed including its arguments.

Examples:

```
mix> stat -l
script_id  process_id  sleep  script
---------  ----------  -----  ------
      u01          -1      0  example1 telnet:sut user1
      u02          23      0  example2 telnet:sut user2
      u03          -1      0  example2 telnet:sut user3
summary: 1/3 running
```

2.3.20 Table

The Mix command `table` builds a mix table from within Mix, which is useful for building large mix tables. The syntax of the `table` command is:

```
table filename arg1 arg2 ... argn
```

The parameter `filename` specifies the name of the new mix table, and the arguments `arg1` through `argn` are entries for the Mix table columns.

The `table` command is most useful when used with a range specification, as shown in the following example:

```
mix> table mix.tab user[01-50], script tty[01-50] user[01-50].1
```

This command would produce the following mix table:

```
user01, script tty01 user01.1
user02, script tty02 user02.1
user03, script tty03 user03.1
...
user50, script tty50 user50.1
```

2.3.21 Time

The `time` command displays the current time.

Example:

```
mix> time
time: 16:09:51
```

2.3.22 Trap

The Mix shell escape command (!) is used to escape Mix temporarily to execute shell commands. The Mix `trap` command specifies an action to be taken if the executed shell command fails, or returns a non-zero exit status.

The syntax of the `trap` command is:

```
trap cmd
```

The parameter `cmd` specifies the action to be taken which can be "exit", "continue", or any valid Mix command (including another shell escape command). For example, when a shell escape command returns a non-zero exit status, Mix will exit if the value of `cmd` is "exit"; or it will continue executing if the value is "continue". The default trap condition is "continue".

The `trap` command can halt execution of Mix, or it can test and reset any function of the emulation environment. This command is useful for executing Mix in batch mode when a shell escape command in the Mix batch file is critical to the execution of the test.

If the shell escape command fails during batch execution, you may not notice an error or be able to end the Mix session before your test is flawed. For example, if the test executes after the shell escape command fails, the false test start could update a database incorrectly which must be rebuilt prior to executing a successful test. By using the trap command set at "exit", you can avoid such an inconvenience.

In the following example Mix session, the trap condition is set to "continue," a shell escape command returns an error, and the Mix session continues:

```
mix> trap continue
mix> use mixtab
mix> set tstart 2
mix> !setupfile
sh: setupfile: not found
mix> start all
```

Now the same example will be executed with the trap condition set to "`exit`":

```
mix> trap exit
mix> use mixtab
mix> set tstart 2
mix> !setupfile
sh: setupfile: not found
$
```

Toggling the trap condition between "continue" and "exit" is sometimes useful during a Mix batch execution. Certain shell escape commands may be critical to test execution and if they malfunction, you should exit the test to make appropriate changes. Other shell escape commands may not be important and the test could continue even if the command did not execute. The following example Mix batch file demonstrates this situation:

```
use mixtab
set tstart 2
set LOGDIR logs
!date
trap exit
!critical_cmd
trap continue
!unimportant_cmd
start all
wait
quit
```

2.3.23 Use

The `use` command identifies the Mix table to be read and used during the test. You must enter this command before a `start` command can execute.

Example:

```
mix> use table1
mix> start all
```

2.3.24 Wait

The `wait` command tells Mix to wait for all scripts to complete before continuing. Upon each script's completion, Mix displays a message containing the current time, the script identifier associated with the script, and the number of scripts still running.

Example:

```
mix> use table1
mix> start all
[user1] started
[user2] started
[user3] started
mix> wait
user1 terminated (2/3 running)
user2 terminated (1/3 running)
user3 terminated (0/3 running)
no more scripts running
mix>
```

2.4 Sample Mix Session

In the following example, two emulated users will run one script each. `User1` will execute the script `example1` and `user2` will execute the script `example2`. The mix table, `table1`, will be used by Mix, and the default log file `mix.log` will be created. During Mix execution, we will instruct Mix to use `table1`, to display script status, to start all scripts, to wait until all scripts have completed, and to quit.

An example of `table1` follows:

```
user1, example1 -n 1 -d serverhost:0
user2, example2 -n 2
```

The example Mix session is presented below:

```
$ mix
Mix: EMPOWER V3.1.8, Serial#R00000-000, Copyright PERFORMIX, Inc. 1988-95

mix> use table1
mix> stat -1
script_id  process_id  sleep  script
---------  ----------  -----  ------
user1      -1          0      example1 -n 1 -d serverhost:0
user2      -1          0      example2 -n 2
summary: 0/2 running
mix> start all
[user1] started
[user2] started
mix> wait
user1 terminated (1/2 running)
user2 terminated (0/2 running)
no more scripts running
mix> quit
$
```

Extract and Report typically are executed after a multi-user emulation. The following example demonstrates the easiest method for executing the Extract and Report commands after the two-user test:

```
$ extract *.1
$ report
```

After executing the Extract and Report commands, your report can be found in the file `report.STD`.

Sections 3.0 Extract and 4.0 Report fully explain how to operate these multi-user tools.

2.5 Distributed Emulations with Mix

Some load tests require multiple UNIX driver machines to emulate large numbers of users. The mix session on your primary UNIX script driver can connect to and control Mix sessions on additional driver machines. The additional mix sessions act as "daemons" receiving instructions from the primary UNIX script driver's mix session.

Mixd, the tool used on run-time (or remote) driver machines, communicates with the master Mix session on the primary UNIX script driver machine to control scripts. The Mix daemons and master Mix session communicate via TCP/IP network connections. All Mix sessions must connect to and communicate through the same TCP port number. The default port number is 7273.

2.5.1 Installing a Run-Time License

After purchasing the Run-Time option, you will need to install the software on each additional UNIX driver machine. To install a Run-Time license, install your EMPOWER software from the distribution media onto each run-time machine. (Clearly, the run-time machine must be binary compatible with the primary UNIX script driver. Refer to Section 3.0 Installation in the Empower Script Development manual for complete installation instructions.) Change to the Install directory created by reading the EMPOWER tape or diskettes. Run `./eminstall`. Follow the instructions given by Eminstall for contacting Performix customer support to obtain a password. Make sure you write "Run-Time" on the password request generated by Eminstall.

When you have your installation password, re-run `./eminstall` and type the password when requested. Eminstall will install the binaries that you read off the tape or diskettes. Only `mixd`, the `gv` commands, and `mon`, (or `xmon` or `csmon` as appropriate) are executable on a Run-Time Licensed machine. You must execute all other tools or commands from the primary UNIX script driver machine.

2.5.2 Methods for Running Multiple Mix Sessions on Remote Machines

After installing Mixd on all of your run-time UNIX driver machines, you may begin a Mix session on a machine in three ways. The first method is simply to type `mixd` at the run-time machine. The Mix daemon will begin and will run in the background, waiting for instructions from the primary UNIX script driver's Mix session.

The second method uses a "mixdaemons" file on the primary UNIX script driver to start remote Mix sessions automatically. This file should be located in the `$EMPOWER` directory and should contain lines with two fields that are separated by commas. The first field is the host name of the run-time machine. The second field is a command that will start the Mix daemon on the run-time machine (typically a remote shell command such as `rsh`, `rcmd`, or `remsh`).

The `connect` command attempts to connect to the Mix daemon on the remote UNIX script driver. Whenever a connection is attempted that is not accepted immediately (i.e., if `mixd` is not running), the "mixdaemons" file will look for a line containing a command to start the Mix daemon. This command then will execute and the connection should be successful.

An example "mixdaemons" file follows:

```
crosby, rsh crosby /usr/empower/bin/mixd
gemini, rsh gemini /usr/empower/bin/mixd
kitfox, rsh kitfox /usr/empower/bin/mixd
```

The third and most advanced method for starting a Mix daemon involves the `inetd` network process daemon. We recommend that you become proficient with the first two methods for running multilple UNIX script drivers before proceeding with the following method.

A Mix daemon will start automatically whenever the master Mix session connects to the run-time machine through the specified TCP port. If you use this method, you must first specify the communications port to be used by adding the following line to the `/etc/services` file (or local variation) on each machine that will run a Mix daemon:

```
mixd 7273/tcp
```

You may change the TCP port number to any number greater than 1024, and this number must be used for all Mix daemons. The default port number may be overridden with the `-p` option of the Mix or Mixd commands (follow the `-p` with the appropriate port number). This option is necessary only if port number 7273 already is used in the `/etc/services` file of any of the UNIX driver machines.

You must add the following line to the `/etc/inetd.conf` file (or local variation) so that the inet daemon runs the correct command when the master Mix session connects to the specified port:

```
mixd\t stream\t tcp\t nowait\t root\t/u/empower/bin/mixd/mixd -I
```

In this line, `\t` represents tab characters. You should replace the directory `/u/empower/bin` with the directory and bin that includes the EMPOWER executables. To ensure that the updated file `/etc/inetd.conf` is used, you must reboot the machine or a `SIGHUP` signal must be sent to the `inetd` process which tells `inetd` to re-read the `inetd.conf` file.

2.5.3 Mix Commands for Distributed Emulations

The following five commands, listed in the Mix help screen, support distributed emulations with Mix (*Note:* `host` represents the name of the run-time UNIX driver machine running a Mix daemon):

Command	Syntax	Description
`connect`	`connect host . . .`	connect to remote Mix daemon(s)
`get`	`get host file . . .`	copy file(s) from host
`put`	`put host file . . .`	copy file(s) to host
`rcd`	`rcd host dir`	change directory on host
`rcmd`	`rcmd host cmd`	run command on host

`connect`	Connects the specified remote Mix daemon(s) to the central Mix session
`get`	Copies the specified file(s) from the specified remote UNIX driver machine to the primary UNIX script driver
`put`	Copies the specified file(s) from the primary UNIX script driver machine to the specified remote UNIX driver machine

(*Note:* The `put` and `get` commands are useful for copying scripts and log files to and from run-time UNIX driver machines.)

`rcd`	Causes the Mix daemon on the specified remote UNIX driver machine to change to the specified directory
`rcmd`	Executes the specified command on the specified UNIX driver machine

Before executing scripts on a remote (run-time) UNIX script driver machine, you should instruct the Mix daemon on that machine (by using the `rcd` command) to change to a directory where you can create files. A common scenario is to "connect" to a run-time machine, "rcd" to a test directory on the machine, "put" the script binaries onto the machine, run the test, and after the test has executed, "get" the log files from the run-time machine.

2.5.4 Executing a Distributed Emulation with Mix

When executing a distributed emulation multiple UNIX script driver machines, you can specify the scripts to be run on each run-time UNIX driver in two ways.

You can include the run-time machine's host name in the Mix table:

```
tinker@rteA,  dbquery telnet:sut tinker
evers@rteB,  dbquery telnet:sut evers
chance@rteC,  dbquery telnet:sut chance
```

Then, start the users on the specified machines by using one of the following Mix `start` commands:

```
$ start  all
$ start  tinker  evers  chance
```

With the second method, you specify the run-time machine's host name in the Mix `start` command as in the following examples:

```
$ start  @rteA  all
$ start  @rteA  tinker  evers  chance
$ start  all@rteA
$ start  tinker@rteA  evers@rteB  chance@rteC
```

Note: You can override a scriptid assigned to the run-time host in the Mix table by specifying a different run-time host in the Mix `start` command. This capability would be useful for redistributing users without editing the Mix table.

You should connect to each run-time Mix daemon explicitly by using the Mix `connect` command at the beginning of the Mix session. Using this command prevents having to restart the entire test if a Mix daemon does not start.

A Mix command file is presented below that includes common steps required to execute a distributed emulation using a run-time UNIX driver machine. This example command file is executed by using the Mix option, `-f filename`.

Our example Mix table follows:

```
user01@hosta, ./script telnet:hostc
user02@hostb, ./script telnet:hostc
user03,  ./script telnet:hostc
user04@hosta, ./script telnet:hostc
user05@hostb, ./script telnet:hostc
user06,  ./script telnet:hostc
```

This table lists entries for starting scripts on two remote UNIX driver machines and on the primary UNIX script driver. (*Note:* Including "./" in script names may be necessary so that the Mix daemon can find the script.)

The `mix.cmd` file follows:

```
# the "use <table>" command will automatically
# connect to all hosts named in the table, but I
# prefer to make the connections explicitly
connect hosta
connect hostb
use table

# remove all old logs in current directory
!rm *.1

# compile the script
!scc script

# change directory to the test directory on hosta and hostb
rcd hosta /usr/testing/test1
rcd hostb /user/testusr/test1

# remove all old logs in the test directories on hosta and hostb
rcmd hosta rm *.1
rcmd hostb rm *.1

# copy the script binary to hosta and hostb
put hosta script
put hostb script
```

(continued on following page . . .)

```
# eminstall the binaries
# eminstall needs to know the value of the EMPOWER variable on the
# run-time machines
rcmd hosta EMPOWER=/usr/empower $EMPOWER/Install/eminstall script
rcmd hostb EMPOWER=/usr/empower $EMPOWER/Install/eminstall script

# start the test
start all

# wait for the scripts to finish
wait

# get the log files from scripts back to the master Mix
get hosta *.1
get hostb *.1
# extract the timestamps from the log files
!extract *.1

# report from the extract files
!report

# done
quit
```

09597594.102600

[This page intentionally left blank]

3.0 Extract

The Extract and Report tools provide response time information required for evaluating the abilities of your system under test (SUT). During script execution, time stamps are placed at the beginning and end of various elements of the log files.

Extract scans for these time stamps to identify the beginning and ending of transactions, functions, and scenarios. Then, Extract retrieves these time stamps saving them in ASCII files called the transaction file, the function file, and the scenario file.

Using these ASCII files, Report calculates response times and produces statistical reports. You also can use your own statistical software package to evaluate the time stamp data retrieved by Extract.

3.1 Response Time

Response time is defined by US government regulations, document (GSA) FPR 1-4.11, as the time that elapses between the transmission of the last character of a command and the receipt of the first printable character of the SUT's response. Transmission of a command typically is completed with activity on the keyboard (or mouse, for an X terminal or PC). A response from the SUT is generally in the form of displayed text or an updated image on your terminal.

On many systems and with many applications, the first printable character of a response often denotes that a transaction is in progress. Messages such as "`busy...`" or "`working...`" appear when transactions on the SUT begin. In such cases, defining the end of the response time as the time at which the first character is received from the SUT may not be desirable. By using this definition, you would determine the time required by the SUT to generate the "`busy...`" message, not the time required by the SUT to complete the work associated with a transaction. With the EMPOWER products, you can mark activities to define the beginning and ending of transactions in a script. Therefore, you can specify response time as the time that elapses between an activity's initiation (by the user) and the completion of some or all work associated with the activity (by the SUT).

3.2 Defining Transactions

You have a great deal of flexibility for defining the beginning and ending of certain activities. In each script file, you should mark locations that make response times generated during a multi-user test correct and meaningful.

Response times can be calculated for three categories of activity: transactions, functions, and scenarios. Functions and scenarios are defined in the script by the following functions: `Beginfunction()`, `Endfunction()`, `Beginscenario()`, and `Endscenario()`. Transactions are defined in EMPOWER and EMPOWER/X scripts by the `Nametransaction()`, `Begintransaction()`, and `Endtransaction()` functions, and in EMPOWER/CS by the `Begintimer()` and `Endtimer()` functions.

Transactions represent individual types of user activities. For example, you may choose to define one or more script interactions as transaction type "login." Script execution will generate a response time for each interaction separately, classifying them as "login" transactions. Report will indicate the average response time for all "login" type transactions.

In EMPOWER/CS scripts, transactions are defined by the `Begintimer()` and `Endtimer()` functions. The word "timer" is used for these functions instead of "transaction" to avoid confusion between true database transactions and the user-level time stamps recorded in a script. In most cases, the term "timer" may be substituted for "transaction" within this documentation. Also, the EMPOWER/CS Monitor and EMPOWER/CS reports will refer to time stamps generated by the `Begintimer()` and `Endtimer()` functions as "transactions."

Functions contain a set of script interactions. The difference between functions and transactions is how the interactions are time-stamped. While individual interactions in a transaction receive time stamps, an entire function receives only one pair of time stamps. Report will indicate the response time for completing all interactions that make up the function. This report will include any think time or type delays that occurred during the set of script interactions.

Scenarios operate essentially the same as functions. Response time associated with a scenario is recorded for the entire set of interactions; however, scenarios typically are used to define an entire user session. For example, during script development, you may define a "word processing" scenario which contains fifteen minutes of activity related to word processing. Generally, a scenario represents an entire script.

3.3 Time Stamp Categories

Time stamps represent the starting and ending times of transactions, functions, and scenarios. Each executing script creates a log file that includes time stamps for each transaction. Additionally, a time stamp is written to a log file every time one of the following functions is encountered in a script: `Beginfunction()`, `Endfunction()`, `Beginscenario()`, or `Endscenario()`

Used only to name transactions in EMPOWER scripts, the `Nametransaction()`, `Begintransaction()`, and `Endtransaction()` functions do not cause time stamps to be written to the log file. EMPOWER automatically recognizes each transaction (i.e., Xmit-Rcv pair) and records one of four types of time stamps in the log file as described below.

The two time stamps associated with transmitting a message from the emulated user are called XT1 and XT2. Usually, keyboard input is considered a transaction. XT1 is the time at which the first character is sent to the SUT; XT2 is the time at which the last character is sent to the SUT. The difference between XT1 and XT2 is the time required to type a command (*Note:* This time is affected by type rate delay).

The time stamps associated with receiving messages from the SUT are called RT1 and RT2. RT1 is the time at which the first character of the response is received. RT2 is the time at which the last character of the response is received. The difference between RT1 and RT2 is the time the SUT requires to send an entire message. The default response time for each transaction is defined as the difference between XT2 and RT2.

For EMPOWER/X, the XT1, RT1, XT2, and RT2 time stamps are recorded only if you insert `Begintransaction()` and `Endtransaction()` functions in the script. These functions must be placed carefully around `KeyString()` functions and their corresponding `Textrcv()` functions.

For EMPOWER/CS, time stamps are recorded in the log file for the `Begintimer()`, `Endtimer()`, `Beginfunction()`, `Endfunction()`, `Beginscenario()`, and `Endscenario()` functions. Transactions are defined in EMPOWER/CS scripts by the `Begintimer()` and `Endtimer()` functions. The word "`timer`" is used instead of "`transaction`" in these functions to avoid confusion between true database transactions and the user-level time stamps recorded in a script. The difference between EMPOWER/CS "`timers`" and EMPOWER "`transactions`" is that `timers` have a begin and end time recorded, while EMPOWER transactions have `Xmit/Rcv` time stamp pairs associated with them (as described in the preceding paragraphs).

Response time for functions and scenarios is calculated as the difference between each time stamp for the `Begin` and `End` functions. For example, if a `Beginfunction()` function executes at 21:28:25.52 and the corresponding `Endfunction()` function executes at 21:28:25.76, then the response time for this function is 0.24 seconds. If a function contains `Think()` functions, response time will include the time the emulated user spends "thinking." `Beginfunction()` and `Endfunction()` functions should be inserted to define specific user activities that will make resulting response time reports meaningful.

3.4 Accounting for Midnight

If you execute an emulation that starts before and ends after midnight, time stamps will be placed in the log files that reflect the actual time of day. However, Extract ensures that these time stamp values do not cause erroneous response time reports.

When Extract scans each log file, it looks for the passing of midnight. If the passing of midnight is detected, Extract will add 24 to the hour value of each subsequent time stamp.

For Report to generate correct throughput results for a multi-user emulation that ran past midnight, Extract must detect midnight in every log file. Therefore, each log file must contain an entry from the first day of the emulation.

3.5 Extract Syntax

The syntax of the `extract` command is shown in the following `extract` usage message. You can access this screen by entering the `extract` command with the parameter "-".

```
$ extract -
EMPOWER V3.1.8, Serial#R00000-000, Copyright PERFORMIX, Inc. 1988-95
Usage:
        extract [-xrsd] [-f filename] [-i scriptlist|script ...]
Options:
        -x                Causes XT1 timestamps to be extracted
        -r                Causes RT1 timestamps to be extracted
        -s                Subtracts timeout times from event times
        -d                Discards events containing timeouts
        -f filename       Names the timestamp files filename.[SFT]
        -i scriptlist     Reads script names from scriptlist file (- is std input)
Notes:
        EXTRACT creates three timestamp files.
        XT2 and RT2 timestamps are extracted by default.
        Data extracted from one log file is stored in script.[SFT].
        Data extracted from multiple log files is stored in extract.[SFT].
        EXTRACT prints the number of remaining log files as they are extracted.
        Script names read from scriptlist file must be newline delimited.
Examples:
        extract script1
        extract script1 script2 script3
        extract -f 3user script1 script2 script3
        extract -r *.1
        extract -i scriptlist
```

3.5.1 Selecting Time Stamps to Extract

As described in Section 3.3, four types of time stamps are produced for EMPOWER and EMPOWER/X scripts that mark transaction response time: XT1, XT2, RT1, and RT2. By default, Extract extracts the time stamps XT2 and RT2 from the log file. Response time generally is considered to be the difference between XT2 and RT2, since this amount represents the time required for the system to respond completely to an input but does not include the time required for the user to make the input.

You can use the -x and -r options of the `extract` command to specify that the time stamps XT1 and RT1 are extracted, respectively. Using one or both of these options, you can override the defaults and will define your own response times.

For example, to extract XT2 and RT1 time stamps (which complies with GSA specifications), you would type the following command:

```
$ extract -r *.1
```

Note that if you specify the `-x` option, the XT1 time stamps are extracted but the XT2 time stamps are not. Likewise, if you specify the `-r` option, the RT1 time stamps are extracted but the RT2 time stamps are not. Under no circumstances are *both* the XT1 and XT2, or RT1 and RT2 time stamps extracted in a single execution of Extract.

3.5.2 Options to Handle Timeouts

Two options of the `extract` command control extraction of time stamps from a log file that contains timeouts. The `-s` option calculates response time for a function containing a timeout by subtracting the timeout time from the extracted function times. The `-d` option discards functions containing timeouts; i.e., the response time for an event containing a timeout is not calculated. (*Note:* These options apply only to scripts using `Beginfunction()` and `Endfunction()` functions.) The `-d` option generally is preferred for log files that contain timeouts.

The `extract` command is executed in the following examples to demonstrate the `-s` and `-d` options:

```
[diane@joelle] extract example2.1
EMPOWER V3.1.8, Serial#R00000-000, Copyright PERFORMIX, Inc. 1988-95

1
warning: can't open example2.1.1
[diane@joelle] extract example2.1
EMPOWER V3.1.8, Serial#R00000-000, Copyright PERFORMIX, Inc. 1988-95

1
[diane@joelle] cat example2.F
15:35:33.43 15:36:05.68 "shell_commands"
[diane@joelle] extract -d example2
```

```
1
[diane@joelle] cat example2.F
15:35:33.43 zz:zz:zz.zz "shell_commands"
[diane@joelle] extract -s example2.1
EMPOWER V3.1.8, Serial#R00000-000, Copyright PERFORMIX, Inc. 1988-95
```

The following EMPOWER log file was created by the script used in the above examples that executes shell commands:

```
>>>         Port: telnet:localhost
>>>         Log: example2.1
>>>         Date: Thu Oct 13 15:35:24 1994
>>>         Command: example2 -d telnet:localhost
EMPOWER V3.1.8, Serial#R00000-000, Copyright PERFORMIX, Inc. 1988-95


>>>         Beginsource("example2")
>>>      22 Set(CDELAY)
>>>      23 Typerate(5.00)
>>>      25 Thinkuniform(1.000,2.500)
>>>      26 Seed(7072)
>>>      27 Timeout(30,CONTINUE)
>>>      28 Unset(NOTIFY)

>>>      29 Signal(USER-DEF)
>>>      32 Term(ZOOM, VT100|LINES24|AUTOWRAP)
>>>      34 Rcv(": ")


>>>         RT1 15:35:24.62
SunOS UNIX (joelle)

login:
>>>         RT2 15:35:24.63
>>>         Think 1.499
>>>         XT1 15:35:25.65
>>>      36 Xmit("empower^M")
>>>         XT2 15:35:27.25
>>>      37 Rcv(":")
>>>         RT1 15:35:27.73
```

(continued on following page ...)

```
empower
Password:
>>>       RT2 15:35:27.83
>>>       Think 2.333
>>>       XT1 15:35:29.84
>>>    39 Xmit("WickleOne^M")
>>>       XT2 15:35:31.89
>>>    40 Rcv("] ")
>>>       RT1 15:35:32.03

Last login: Thu Oct 13 15:34:45 from LOCALHOST.perfor
SunOS Release 4.1.2 (GENERIC) #2: Wed Oct 23 10:52:58 PDT 1991

MODEM USERS:  att1:tty22 dialout at up to 2400.
              att1:tty24 dialout at up to 9600.
              joelle:ttya dialin at up to 9600 (703-760-9241).

120MB tape     /dev/rmt0
3 1/2 floppy   /dev/fd0         (fdformat to floppy)

run openwin if you want OpenWindows
inc: no mail to incorporate
[empower@joelle]
>>>       RT2 15:35:33.43
>>>    44 Beginscenario("example2") 15:35:33.43
>>>    46 Beginfunction("shell_commands") 15:35:33.43
>>>       Think 1.893
>>>       XT1 15:35:34.44
>>>    51 Xmit("date^M")
>>>       XT2 15:35:35.47
>>>    52 Rcv("% ")
>>>       RT1 15:35:35.53
date
Thu Oct 13 15:35:35 EDT 1994
[empower@joelle]
>>>       Timeout 15:36:05.67
>>>    57 Endfunction("shell_commands") 15:36:05.68
>>>       Think 1.220


>>>       XT1 15:36:06.70
>>>    58 Xmit("exit^M")
>>>       XT2 15:36:07.76
```

(continued on following page ...)

```
>>>     59 Wait(2)
>>>        RT1 15:36:07.78
exit
>>>        RT2 15:36:07.78
>>>     61 Endscenario("example2") 15:36:07.92
>>>        Endsource()
>>>        Closed: telnet port
```

3.5.3 Redirecting Extract Output

When Extract reads a script log file for time stamp data, it places the data in, if appropriate, a transaction file ending with ".T", a function file ending with ".F", and a scenario file ending with ".S". If Extract reads only one log file, the prefix of these files will be the same prefix of the file that Extract read.

Example:

```
$ extract  example1
```

The above example would create files called `example1.T`, `example1.S`, and `example1.F`. When Extract scans more than one log file for time stamp data, it places the time stamp data in files called `extract.T`, `extract.S`, and `extract.F`.

In either case, if you want Extract to place the time stamp data in files with a different prefix, you can do so with the `-f` option of the `extract` command. The parameter of the `-f` option is the prefix name of the time stamp data files. The following command will place Extract output in files called `1user.T`, `1user.S`, and `1user.F`:

```
$ extract  -f  1user  example1
```

3.5.4 Log File Specification

Each user in a multi-user emulation creates a separate log file. When you execute Extract, you must specify the log files to be extracted in one of two ways.

One method for specifying log files is with the `-i` option. With this option, you specify either a file name or a hyphen as a parameter. The file name specifies a file containing a list of all log files to be extracted. The hyphen specifies that Extract will receive the list of log files from standard input—typically this means that you enter the log file names at the UNIX script driver machine.

For example, if the file `loglist` contains a list of log files to be extracted, the following `extract` command will use that list:

```
$ extract  -i  loglist
```

If you wish to enter the log file names at the UNIX script driver, use the following `extract` command:

```
$ extract  -i  -
```

The most common method for specifying log files is to specify file names directly in the `extract` command. The `extract` command's `script` parameter is the name, or prefix, of the log file that Extract will scan for time stamp data. Extract adds a .l extension to the name specified. If you gave your log file a name other than the one given by default during script execution, make sure that you specify the correct log file name. You may specify multiple log files in one `extract` command—simply list all log file names as parameters.

Examples:

```
$ extract  example1  example2  example3
$ extract  example1.1  example2.1  example3.1
```

You also may use the wild card character * to specify multiple files.

```
$ extract  example*.l
$ extract  user*.l
$ extract  *.l
```

Note that Extract will recognize the .l extension and will not try to add an additional extension. In the last example (`extract *.l`, often used in multi-user load tests), Extract will use all files in the current directory that have the .l extension.

When Extract scans multiple files, it will display a number on the UNIX script driver that indicates the remaining number of log files to be scanned. This display is helpful when you have used the command `extract *.l` to scan all files with the .l extension. If you are running a ten user test, you should see the number "10" appear on the UNIX script driver when Extract executes. As each file is read, new numbers appear, counting down until all files have been read. The following example demonstrates this process after all files have been scanned by Extract:

```
$ extract  *.l
Extract: EMPOWER V3.1.8, Serial#R00000-000, Copyright PERFORMIX, Inc.
1988-95

10  9  8  7  6  5  4  3  2  1
$
```

Note: If you execute Extract for a ten user test and the first number displayed is "11," you probably need to delete an old log file.

3.6 Sample Extract Execution

In the following example, Extract scans the log file called `example1.1`:

```
$ extract  example1
Extract: EMPOWER V3.1.8, Serial#R00000-000, Copyright PERFORMIX, Inc.
1988-95

1    ,
$
```

The number of log files to be scanned appears below the Extract Copyright statement. In this example the number is 1. Extract places time stamp data in the files `example1.T`, `example1.F`, and `example1.S`. The format of a time stamp is `hh:mm:ss.xx` where `hh` is the hour, `mm` is the minute, `ss` is the second, and `xx` is the hundredth of a second.

Assuming that we had added the appropriate `Nametransaction()` or `Begintransaction()` and `Endtransaction()` functions in the script, the file `example1.T` might look like this:

```
10:56:53.20 10:56:53.48 "date"
10:56:56.40 10:56:52.01 "ls"
```

If not, it would look like this:

```
10:56:53.20 10:56:53.48 " "
10:56:56.40 10:56:52.01 " "
```

If we had specified the appropriate `Beginfunction()` or `Endfunction()` functions, `example1.F` might contain the following records:

```
10:56:02.11 10:56:20.56 "query_database"
10:56:20.56 10:56:22.66 "enter_data"
10:56:30.70 10:56:50.08 "vi_session"
10:56:52.21 10:57:02.23 "quit_database"
```

If we hadn't specified `Beginfunction()` or `Endfunction()` functions, the file would be empty.

Because we extracted from only one log file, `example1.s` would contain only two records: the number of log files scanned and time stamps for the beginning and end of the scenario. If we had extracted more logs, each log would have a beginning and end scenario time.

```
1
10:55:54.83 10:57:02.40 "example1"
```

4.0 Report

The Report tool reads Extract's output to generate response time and throughput data. This information can be presented in standard transaction, function, and scenario reports or in a Government Services Administration (GSA) compliant report. These reports are stored in two files: one with a .STD extension, and one with a .GSA extension. The standard report is produced by default; the GSA report is generated with the -G option of the report command.

4.1 Report Syntax

The following usage message lists the syntax of the report command. You can access this menu by typing the report command with a hyphen:

```
$ report -
EMPOWER V3.1.8, Serial#R00000-000, Copyright PERFORMIX, Inc. 1988-95
Usage:  report [-BEFSThmdc12345678] [-b t] [-e t] [-f f] [-p n]
        [-u n] [-w n] [-s n] [script]
Options:
        -b t    Changes begin time of sample to t
        -e t    Changes end time of sample to t
                Default includes events ending in sample
        -B      Includes only events starting in sample
        -BE     Includes only events starting and ending in sample
        -F      Causes a function report to be produced
        -G      Causes a GSA report to be produced
        -S      Causes a scenario report to be produced
        -T      Causes a transaction report to be produced
        -h      Suppresses headings
        -m      Changes report units to minutes (default is seconds)
        -d      Forces within values to be discrete, not cumulative
        -c      Supresses default columns
        -[1-8]  Forces default column n to appear
        -f f    Changes the output file names to f.GSA and f.STD
        -p n    Computes the nth response time percentile
        -u n    Changes the number of users to n
        -w n    Computes number of events completed within n units
        -s n    Changes the size of .STD name fields to n (default: 14)
```

0595754" 102600

4.1.1 Specifying a Begin and End Time

Report will calculate the duration of each emulated user's session which is used to calculate the system under test's (SUT's) throughput. The duration is the difference between the begin time and the end time for each report.

By default, the begin time is when the first `Beginscenario()` executes and the end time is when the last `Endscenario()` executes. However, Report will calculate statistics for all interactions in the Extract output file, including interactions that occurred before the first `Beginscenario()` execution or after the last `Endscenario()` execution. Therefore, if the default begin and end times are used, activities that occur before the first `Beginscenario()` execution or after the last `Endscenario()` execution will cause erroneous results.

To create reports that calculate data for a specified time other than the default begin and end times, use the `-b` and `-e` options of the `report` command. The `-b` option specifies the report start time; the `-e` option specifies the report end time. The time is specified as hours (`hh`), hours and minutes (`hh:mm`), or hours, minutes, and seconds (`hh:mm:ss`).

You also may use the key words `first` and `last` with the `-b` and `-e` options. The option `-b first` specifies that the begin time is the time of the first `Beginscenario()` execution. The option `-b last` specifies that the begin time is the time of the last `Beginscenario()` execution.

The option `-e first` specifies that the end time is the time of the first `Endscenario()` execution. The option `-e last` specifies that the end time is the time of the last `Endscenario()` execution. The following examples demonstrate the -b and -e commands:

```
$ report -b 10:20 example1

$ report -b 10:20 -e 11:20 example1

$ report -e last example1
```

4.1.2 Specifying Events for a Time Range

By default, report will include only the interactions that end in the specified time sample which ensures that an interaction is counted only once if it spanned two specified time samples. If you want to override the default and include the interactions that began within a specified range, use the `-B` option in the `report` command. Specify the `-BE` option if you want to include interactions that began and ended within a specified range.

The following example `report` commands use the `-b`, `-e`, `-B`, and `-E` options. These examples will include interactions that began and ended between 10:20 and 11:20:

```
$ report -b 10:20 -e 11:20 -BE example1

$ report -beBE 10:20 11:20 example1
```

This last example includes interactions that began between 10:20 and 12:20:

```
$ report -TppwbeB 80 90 2.5 10:20 12:20 example1
```

4.1.3 Report Selection

The `-S`, `-F`, and `-T` options specify how the standard report will be generated. The `-S` option will include a scenario report, the `-F` option will include a function report, and the `-T` option will include a transaction report. You can select one, two, or all three of the options. If none of these options are selected, all three reports will be generated by default. If the script does not include `Beginfunction()` and `Endfunction()` functions in the script, producing a function report will not be possible.

4.1.4 Generating a GSA Compliant Report

By default, Report generates a standard (`.STD`) report. If you need to generate the GSA compliant report, use the `-G` option of the `report` command:

```
$ report -G
```

4.1.5 Suppressing Report Headers

By default, the STD report file will contain header information identifying column titles. If you do not want the header information included (perhaps to process report data with other UNIX commands) specify the `-h` option with the `report` command.

4.1.6 Unit Selection

By default, response times are reported in seconds. If your response times are very large, which may be the case with response times of compilations or data base sorts, then you may wish to change the unit of response time reporting to minutes. The `-m` option of the `report` command will cause Report to display response times in minutes.

4.1.7 Within Value Selection

Report has the ability to identify the number of events that completed within a specified amount of time. For example, you may want to know how many transactions completed within two seconds or less.

To select a "within" value, use the `-w` option with a floating point number identifying the time. You can string a set of `-w` options together and list all of the times. In the following examples, Report will generate a transaction report (`-T`) identifying the number of transactions that completed within (`-w`) 1 second, 2.5 seconds, and 5 seconds.

```
$ report -T -w 1 -w 2.5 -w 5 example1

$ report -Twww 1 2.5 5 example1
```

The `-w` option specifies a cumulative report meaning that for each "within" value specified, the report shows the*total* number of events completed within the specified amount of time. The `-d` option of the `report` command reports discreet "within" values. When multiple "within" values are specified, Report indicates the incremental number of events completed within the specified amount of time.

For the above example, suppose:

- five events were completed within one second,
- three more events completed between one and two and a half seconds, and
- seven more events completed between two and a half and five seconds.

If the `-w` option is used to generate the report, the report will indicate that for the report at:

- one second, five events had completed
- two and a half seconds, eight events had completed
- five seconds, fifteen events had completed

If the `-d` option also is used to generate the report, the report will indicate that for the report at:

- one second, five events had completed
- two and a half seconds, three events had completed
- five seconds, seven events had completed

4.1.8 Suppressing and Forcing Report Columns

Report generates a standard report which includes a set of nine reporting columns. These columns are: Interaction Name (i.e., Scenario, Function, or Transaction), Total, Finish, Thruput, Median, Average, Minimum, Maximum, and Std-Dev. With the `-c` and `-[1-8]` options of the `report` command, you can generate a customized reporting format with some or all of these columns.

The `-c` option suppresses the eight statistical columns; the Interaction Name column can not be suppressed. The `-[1-8]` options specify columns to be displayed by number. For example, suppose you want to generate a customized version of the standard report in which you show only the Interaction Name, Total, Finish, Average, Min, Max, and Std-Dev columns. In addition, you want to show the 75th and 95th percentiles. In the `report` command, you would first suppress all columns with the `-c` option, then specify individual columns for display. The `-p` option specifies percentiles and the `-f` option specifies the name of the report file:

```
$ report -c -125678 -pp 75 95 -f custom
```

This command creates the following report, `custom.STD`:

```
EMPOWER Standard Report

Date:        Wed Apr  7 10:52:55 1993
Start time: 10:41:32
Stop time:  10:42:26
Duration:   00:00:54
Mix:        4 users
Unit:       seconds

Scenario    Total  Finish Average Minimum Maximum Std-Dev    P75      P95
-------------------------------------------------------------------------
script1         4       4   48.91   43.79   53.97    4.15  51.86    53.97
-------------------------------------------------------------------------

Function    Total  Finish Average Minimum Maximum Std-Dev    P75      P95
-------------------------------------------------------------------------
word_proc       4       4    3.16    2.62    3.80    0.42   3.19     3.80
db_query        4       4    3.54    2.34    5.71    1.30   3.38     5.71

Overall         8       8    3.36    2.34    5.71    0.99   3.38     5.71
-------------------------------------------------------------------------

Transaction Total  Finish Average Minimum Maximum Std-Dev    P75      P95
-------------------------------------------------------------------------
date           20      20    0.24    0.10    0.45    0.09   0.28     0.35
loginid         4       4    0.18    0.12    0.33    0.08   0.16     0.33
logout          4       4    0.02    0.01    0.02    0.00   0.02     0.02
ls             20      20    0.28    0.11    0.68    0.13   0.32     0.45
passwd          4       4    5.00    2.58    6.95    1.61   5.80     6.95
pwd             4       4    0.05    0.02    0.13    0.05   0.02     0.13
termtype        4       4    2.42    1.90    3.72    0.76   2.12     3.72
who            20      20    0.26    0.10    0.55    0.11   0.33     0.43

Overall        80      80    0.58    0.01    6.95    1.20   0.33     2.58
-------------------------------------------------------------------------
```

4.1.9 Redirecting Report Output

By default, Report places output in a file ending with the suffix, `.STD`. The prefix is obtained from the Extract output files used to generate reports.

If you want Report to place output in a file with a different prefix, you can use the -f option of the report command.

The following command will place output in luser.STD:

```
$ report -f luser example1
```

4.1.10 Percentile Selection

Response times can be reported by breaking measured transactions into response time percentiles. The -p option of the report command specifies that response time percentiles should be calculated and reported. You should enter an integer value between one and 100 after the -p option. As shown in the following examples, you may set a series of response time percentiles in several ways. These examples specify that a transaction report will generate with the 70th, 80th, 90th, 95th, and 99th percentile response times. Remember that the -c option is used to suppress the standard report columns.

```
$ report -c -T -p 70 -p 80 -p 90 -p 95 -p 99 example1

$ report -c -T -ppppp 70 80 90 95 99 example1

$ report -c -Tppppp 70 80 90 95 99 example1
```

The example report, `example1.STD` follows:

```
EMPOWER Standard Report

Date:           Mon Oct 17 16:26:59 1994
Start time:     16:23:44
Stop time:      16:25:37
Duration:       00:01:53
Mix:            10 users
Unit:           seconds

Transaction          P70     P80     P90     P95     P99
------------------------------------------------------
date                0.50    0.69    1.04    1.14    1.65
login               2.04    2.45    2.97    3.22    3.35
logout              0.02    0.02    0.04    0.04    0.04
ls                  2.60    2.68    3.06    3.21    3.79
ps                  5.78    5.87    6.56    7.72    8.72
who                 1.57    1.94    2.46    3.15    3.75

Overall             2.53    3.12    4.84    5.83    7.72
------------------------------------------------------
```

A percentile is defined as a value below which a certain percentage of the observations fall. In this report, 90 percent of all transactions occurred in 4.84 seconds or less, and 90 percent of the date transactions occurred in 1.04 seconds or less.

Note: If the `-p` option is not used, response time percentiles will not be reported.

4.1.11 Specifying Number of Users

By default, Report uses the number of log files as the number of users, which is accurate if each emulated user creates one log file. If the number of emulated users is not the same as the number of log files created, you can change the number of users with the `-u` option of the `report` command The following command will change the number of users to 32.

```
$ report -u 32 example1
```

4.1.12 Changing the Field Name Size

The field size in a standard report specifying the transaction, function, or scenario is 14 characters wide by default. With the `-s` option, you can change the field size to a minimum of 11 characters and a maximum of 100 characters. For more column space, use the `-s` option to specify a smaller field size.

4.1.13 Script File Specification

The report command's `script` parameter identifies the script that produced the log from which time stamp data was extracted. Report will add a `.S` extension to the file name when searching for the extracted scenario file, a `.F` extension when searching for the extracted function file, and a `.T` extension when searching for the extracted transaction file.

If you do not specify a script, Report looks for the files `extract.S`, `extract.F`, and `extract.T`. Recall that Extract writes to `extract.S`, `extract.F`, and `extract.T` when it extracts from more than one log file.

4.2 Report Environment Variables

In addition to configuring reports with command line options, you can configure them with environment variables. Report understands three environment variables.

The first environment variable is `E_VENDOR`. If `E_VENDOR` is defined, Report will use its value in the GSA and STD reports as the name of the vendor for whom the report is being prepared. In the example reports shown in the following sections, `E_VENDOR` is defined as "`Your Company Name`".

The second and third environment variables are `E_ACTIVITY` and `E_PROJECT`. If either of these variables are defined, their values also will be displayed below the title of the GSA and STD reports. `E_ACTIVITY` and `E_PROJECT` typically are used to identify SUT configuration and user load on the UNIX driver machine associated with the report.

4.3 Sample Report Execution

GSA and Standard reports can be viewed with UNIX commands such as `more`, `cat`, and `pg`. Examples of these reports follow.

4.3.1 The GSA Report

Contents of example1.GSA - Part 1

```
                    EMPOWER Scenario Summary Report

Project:                                10 User Test
Vendor:                                 Your Company Name
Date:                                   Tue Oct 18 14:31:47 1994
Activity:                               Your SUT Configuration

Summary for scenario:                   Overall

Start time:                             16:23:44
Stop time:                              16:25:37
Duration:                               00:01:53
Number of scenarios attempted:          10
Number of scenarios completed:          10
Completion rate:                        0.09 scenarios per second

Median time:                            92.75 seconds
Average time:                           93.78
Minimum time:                           85.68
Maximum time:                           101.18
Standard deviation:                     4.14

Summary for scenario:                   "example1"

Start time:                             16:23:44
Stop time:                              16:25:37
Duration:                               00:01:53
Number of scenarios attempted:          10
Number of scenarios completed:          10
Completion rate:                        0.09 scenarios per second

Median time:                            92.75 seconds
Average time:                           93.78
Minimum time:                           85.68
Maximum time:                           101.18
Standard deviation:                     4.14
```

Contents of example1.GSA - Part 2

```
              EMPOWER Interactive Response Time Summary Report

Project:                                10 User Test
Vendor:                                 Your Company Name
Date:                                   Tue Oct 18 14:31:47 1994
Activity:                               Your SUT Configuration
Summary for transaction:                Overall

Start time:                             16:23:44
Stop time:                              16:25:37
Duration:                               00:01:53
Number of transactions attempted:       230
Number of transactions completed:       230
Completion rate:                        2.03 transactions per second

Median time:                            1.29 seconds
Average time:                           1.90
Minimum time:                           0.00
Maximum time:                           8.72
Standard deviation:                     1.89

Summary for transaction:                Unspecified

Start time:                             16:23:44
Stop time:                              16:25:37
Duration:                               00:01:53
Number of transactions attempted:       180
Number of transactions completed:       180
Completion rate:                        1.59 transactions per second

Median time:                            1.58 seconds
Average time:                           2.10
Minimum time:                           0.00
Maximum time:                           8.72
Standard deviation:                     2.03

Summary for transaction:                "who"

Start time:                             16:23:44
Stop time:                              16:25:37
Duration:                               00:01:53
Number of transactions attempted:       50
Number of transactions completed:       50
Completion rate:                        0.44 transactions per second

Median time:                            0.91 seconds
Average time:                           1.18
Minimum time:                           0.28
Maximum time:                           3.75
Standard deviation:                     0.92
```

The GSA report is formatted to meet US Government GSA standards. Separate GSA reports are prepared for each type of interaction measured (transaction, function, and scenario) if the interaction has at least one occurrence in the report period. For each type of interaction, the following reports may be generated:

- Overall Summary—a summary of all occurrences of the interaction
- Unspecified Summary—a summary of unnamed interactions
- Named Interaction Summary—a summary of named interactions

Note: Unspecified transactions are transmit-receive pairs that are not named with `Nametransaction()` or `Begintransaction()` and `Endtransaction()`. Unspecified functions and scenarios would result only if the string in the `Beginfunction()/Endfunction()` or `Beginscenario()/Endscenario()` functions was null (e.g., `Beginfunction("")`). However, you rarely would use these functions in such a way.

Each GSA report includes a title, the report date, and start and stop times of the extracted data.

4.3.1.1 Start Time and Stop Time

The start time for each report is the earliest moment at which a `Beginscenario()` function was executed. The stop time is the moment at which the last `Endscenario()` function completed. The duration is the difference between the start and stop times. The default start and stop times are overridden by the `-b` and `-e` report options as described in section 4.1.1.

4.3.1.2 Events Attempted and Completed

The GSA format also displays the number of events attempted and completed. Any interaction that begins during script execution is an attempt, even if it did not end. For example, the execution of a `Beginfunction()` function without the execution of the corresponding `Endfunction()` function is an incomplete attempt.

4.3.1.3 Completion Rate

The completion rate is computed as the number of completed events divided by the duration of the emulation. This completion rate identifies*throughput* of the SUT.

4.3.1.4 Median

The median is the middle value, or the arithmetic mean of two middle values, in a distribution. In this case, median refers to the response time value that half of the interaction response times are greater than and half are less than.

4.3.1.5 Average

The average is the arithmetic mean of a distribution. The average interaction response time is the sum of the response times divided by the number of interactions.

4.3.1.6 Minimum

The minimum is the smallest value in a distribution. In our reports, minimum refers to the minimum interaction response time value.

4.3.1.7 Maximum

The maximum is the largest value in a distribution. In our reports, maximum refers to the maximum interaction response time value.

4.3.1.8 Standard Deviation

Standard deviation is a measure of dispersion in a distribution. It is defined as the square root of the arithmetic average of the squares of deviations from the mean. For our reports, standard deviation is used in conjuction with the average to help determine the amount that each interaction's response time varies from the average.

Note: Using percentiles often is easier for obtaining information on interaction response time distribution.

For more information on these statistical measurements, you can refer to any statistical reference text.

4.3.2 The Standard Report

An example of a typical EMPOWER standard report follows:

Contents of example1.STD

```
EMPOWER Standard Report

Project:         10 User Test
Vendor:          Your Company Name
Date:            Tue Oct 18 16:29:48 1994
Activity:        Your SUT Configuration
Start time:      16:23:44
Stop time:       16:25:37
Duration:        00:01:53
Mix:             10 users
Unit:            seconds

Scenario      Total  Finish Thruput  Median Average Minimum Maximum Std-Dev
---------------------------------------------------------------------------
example1         10      10    0.09   92.75   93.78   85.68  101.18    4.14
---------------------------------------------------------------------------

Transaction   Total  Finish Thruput  Median Average Minimum Maximum Std-Dev
---------------------------------------------------------------------------
who              50      50    0.44    0.91    1.18    0.28    3.75    0.92
Unspecified     180     180    1.59    1.58    2.10    0.00    8.72    2.03

Overall         230     230    2.03    1.29    1.90    0.00    8.72    1.89
---------------------------------------------------------------------------
```

The STD report contains the same information as the GSA report in a more condensed format. This report displays summary information for each event (interaction) type. See Section 4.3.1 above.

Like the GSA format, these summaries include information on specific events as well as overall and unspecified event summaries. The report columns identify the total number of attempted events; the number of completed events; the event's throughput; the event's median, average, minimum, and maximum response times; and the standard deviation of response times. (*Note:* The unspecified event category applies only to EMPOWER, not to EMPOWER/X or EMPOWER/CS)

[This page intentionally left blank]

5.0 Draw

Using one or more standard Report files, the Draw tool creates bar charts that compare results of a multi-user test. Draw can be used to compare statistics of several transaction types for a single user emulation or to compare several multi-user emulations. This tool will generate bar charts that contain up to 60 events found in a single standard report and also will compare a single transaction, scenario, or function for up to 60 different multi-user emulations.

Draw provides an added dimension to Report results. The output of Report answers such questions about a single user emulation as:

- "What was my average response time during the 16-user run?"
- "How many data base updates can the system complete in an hour when 100 users are active?"

The output of Draw answers questions about one or more multi-user emulations such as:

- "How does my average response time change as the workload increases from 10 users to 100 users?"
- "How does the 80th response time percentile look for each function in the 16 user run?"

Draw produces bar charts, or histograms, that can be viewed on-line or printed on any system printer. The output of Draw is in ASCII format and contains no printer-specific characters.

The following figure shows a sample Draw output that identifies the change in response time as the number of users on a system is increased. After eight users, the system under test was saturated, forcing the response time for an additional user load to increase significantly.

Sample Draw Output—Response Time Vs. Number of Users

```
                              EMPOWER Bar Chart
                              -----------------
    Function:                                           Legend:
         query                                              @ - Average

       7.5 +- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
   R       |
   e       |
   s       |
   p       |                                                              @
   o   6.0 +- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  @ -
   n       |                                                              @
   s       |                                                              @
   e       |                                                              @
           |                                                              @
   T   4.5 +- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  @ -
   i       |                                                              @
   m       |                                                              @
   e       |                                                              @
           |                                                              @
   s   3.0 +- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  @ -
   e       |                              @                               @
   c       |                              @                               @
   o       |                              @                               @
   n       |                              @                               @
   d   1.5 +- - - - - - -@- - - - - - - -@- - - - - - - - - - - - - - -  @ -
   s       |@   @        @                @                               @
           |@   @        @                @                               @
           |@   @        @                @                               @
           |@   @        @                @                               @
        .0 +@---@--------@----------------@-------------------------------@--
            1   2        4                8                               1
                                                                          6

                         N u m b e r   o f   u s e r s

              +---------------------------------------------------+
              |   Number of users      |  Response Time seconds   |
              +------------------------+--------------------------+
              |Users|    Input file    |         Average          |
              +-----+------------------+--------------------------+
              |  1  |  1user.STD       |          1.14            |
              |  2  |  2user.STD       |          1.19            |
              |  4  |  4user.STD       |          1.55            |
              |  8  |  8user.STD       |          2.61            |
              | 16  |  16user.STD      |          6.13            |
              +---------------------------------------------------+
```

5.1 Batch Mode Execution

Draw can be executed in batch mode if a file exists that specifies the chart's format. The following example shows a bar chart's specification file that will compare minimum, maximum, and average response time of "query" and "update" transactions during a multi-user run.

```
BEGIN
TITLE = EMPOWER Bar Chart
XTITLE = Transaction
YTITLE = Response Time seconds
INPUT = 16user.STD
EVENT = Transaction
X = query, update
Y = Average, Minimum, Maximum
LEGEND = @, |, *
ORGANIZE = CLUSTERED
YMIN = 0.0
YMAX = 20.
COMMENT = Created for ABC Corp
COMMENT = January 2, 1991
END
```

If this file is called `draw.spec`, then the following command will execute Draw in batch mode. (*Note:* The complete syntax for the `draw` command is discussed in Section 5.4).

```
$ draw -s draw.spec -o draw.out
Draw: EMPOWER V3.1.8, Serial#R00000-000, Copyright PERFORMIX, Inc. 1988-95
Bar chart(s) drawn.
Draw exited.
$
```

As directed by the specification file, Draw would create a bar chart that places response time on the Y-axis and transaction types on the X-axis. Response times of the "update" transactions are noticeably larger than response times for the "query" transactions.

Sample Draw Output for Batch Mode Execution

```
                            EMPOWER Bar Chart
                            -----------------
  Start time  20:27:01                       Legend:
  Stop time   20:32:42                            @ - Average
  Duration    341.06 seconds                      | - Minimum
  Mix         16 users                            * - Maximum
  Input file  16user.STD

   20.0 +- - - - - - - - - - - - - - - - - - - - - - - - - -*- - - -
R       |                                                   *
e       |                                                   *
s       |                                                   *
p       |                    *                              *
o  16.0 +- - - - - - - - - - * - - - - - - - - - - - - - - -*- - - -
n       |                    *                              *
s       |                    *                              *
e       |                    *                              *
        |                    *                              *
T  12.0 +- - - - - - - - - - * - - - - - - - - - - - - - - -*- - - -
i       |                    *                              *
m       |                    *                              *
e       |                    *                              *
        |                    *                            @ *
s   8.0 +- - - - - - - - - - * - - - - - - - - - - - - - -@-*- - - -
e       |                    *                            @ *
c       |                  @ *                            @ *
o       |                  @ *                            @ *
n       |                  @ *                            @ *
d   4.0 +- - - - - - - - - @ * - - - - - - - - - - - - - -@-*- - - -
s       |                  @ *                            @ *
        |                  @ *                            @ *
        |                  @ *                            @ *
        |                  @|*                            @|*
     .0 +------------------@|*----------------------------@|*--------
                               A                              B
                         T r a n s a c t i o n s
  +------------------------+-----------------------------------------+
  |       Transaction      |            Response time seconds        |
  +-----+------------------+-------------+--------------+------------+
  | ID  |    Description   |   Average   |   Minimum    |  Maximum   |
  +-----+------------------+-------------+--------------+------------+
  |  A  | query            |    6.13     |     0.89     |   16.35    |
  |  B  | update           |    8.87     |     0.61     |   19.98    |
  +-----+------------------+-------------+--------------+------------+
```

5.2 Interactive Mode Execution

Specifications for bar charts generally are created interactively. To enter interactive mode, change to the working directory where your standard report files are stored and enter the `draw` command without any options. The working directory must contain some standard report files or Draw will display an error and terminate.

The following example shows an interactive session that will produce the specification and chart shown in section 5.1. In this example, CR indicates pressing the carriage return key:

```
$ draw
Draw: EMPOWER V3.1.8, Serial#R00000-000, Copyright PERFORMIX, Inc. 1988-95

Please respond after each => prompt.
Press RETURN to accept a default.
Type quit at any time to exit.

Please select one or more INPUT file names.
Available files in the current directory are:
  1) 16user.STD   2) 1user.STD    3) 2user.STD
  4) 4user.STD    5) 8user.STD
Enter * to select all the files.
=> 1
Please select an EVENT to be charted.
  1) Scenario          2) Function          3) Transaction
Default is 1) Scenario
=> 3
Please select one or more Transactions for the X axis.
Available X axis values are:
  1) Overall      2) Unspecified      3) query      4) update
Default is 1) Overall
=> 3 4
Please select one or more statistics for the Y axis.
Available Y axis values are:
  1) Total       2) Finish       3) Thruput      4) Median
  5) Average     6) Minimum      7) Maximum      8) Std-Dev
Default is 5) Average
=> 5 6 7
Please enter 3 LEGEND characters.
Default characters are @ | *
=> CR
```

(continued on following page . . .)

```
Please select an ORGANIZE value.
  1) CLUSTERED      2) HIDDEN
Default is 1) CLUSTERED
=> CR
Please enter YMIN and YMAX values.
Defaults are 0.0 and 20.
=> CR
Please enter the TITLE of the bar chart.
Default is EMPOWER Bar Chart
=> CR
Please enter the X axis TITLE.
Default is Transaction
=> CR
Please enter the Y axis TITLE.
Default is Response Time seconds
=> CR
Please enter comments to be included in the bar chart.
Type a blank line to end.
=> Created for ABC Corp
=> February 25, 1990
=> CR

Spec for bar chart is:
  BEGIN
  TITLE = EMPOWER Bar Chart
  XTITLE = Transaction
  YTITLE = Response Time seconds
  INPUT = 16user.STD
  EVENT = Transaction
  X = query, update
  Y = Average, Minimum, Maximum
  LEGEND = @, |, *
  ORGANIZE = CLUSTERED
  YMIN = 0.0
  YMAX = 20.
  COMMENT = Created for ABC Corp
  COMMENT = February 25, 1990
  END
Would you like to save this spec (y/n)?
Default is y) yes
=> CR
Please enter name of spec file.
Default is draw.spec
=> CR
draw.spec exists.
Do you want to append or overwrite (a/o)?
Default is o) overwrite
=> CR
```

(continued on following page . . .)

```
.
Do you want to create another spec (y/n)?
Default is n) no
=> CR
Do you want to create the bar chart (y/n)?
Default is y) yes
=> CR
Please enter name of output file.
Default is draw.out
=> CR
draw.out exists.
Do you want to append or overwrite (a/o)?
Default is o) overwrite
=> CR
1 bar chart written to draw.out
Draw: normal completion.
```

During this interactive session, a specification file called `draw.spec` was created and the bar chart created was stored in `draw.out`. These two files are identical to the files used in Section 5.1.

5.3 Bar Chart Format

Every bar chart created by Draw, interactively or in batch mode, is composed of four sections: header, figure, data table, and messages. If it contains 14 or fewer bars or clusters, the entire chart will fit on one standard 8-1/2 x 11 page . If there are more than 14 bars or clusters, the data table will continue to the next page.

5.3.1 Header

The header is the top portion of the chart. It consists of a centered title at the top, a legend on the right, and chart information on the left. The chart information displayed depends on the number of specified INPUT files. If only one INPUT file is specified, the start time, stop time, duration, number of users, and name of the INPUT file will be displayed . If multiple INPUT files are specified, the name of the transaction, scenario, or function that is charted will be displayed .

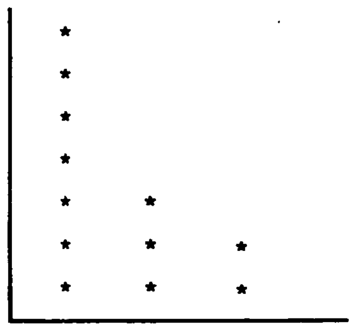
5.3.2 Figure

The figure is displayed below the header. The range of the vertical axis is defined by the YMIN and YMAX values in the specification. Draw may fine tune your values to make the chart more visually appealing. The horizontal axis identifies each bar with an ID depending on the number of INPUT files specified. If one INPUT file is specified, the ID will be a character related to the data table information. If multiple files are specified, the ID will be the number of users in the corresponding INPUT file .

Each bar is composed of one to three LEGEND characters depending on the Y and ORGANIZE fields in the specification. Every performance measurement specified in the Y field is associated with a specific LEGEND character.

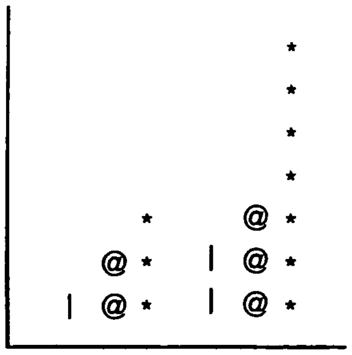
If one performance measurement is specified for the Y field, then the bars will be evenly spaced and will contain the same LEGEND character.

Example:



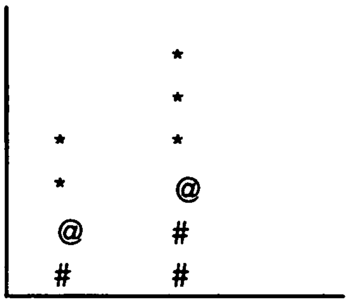
If two or three performance measurements are specified for the Y field and the ORGANIZE value is CLUSTERED, each bar will contain its specific LEGEND character and the bars will be grouped together in a cluster.

Example:



If two or three performance measurements are specified for the Y field and the ORGANIZE value is HIDDEN, each bar will contain two or three LEGEND characters. The taller bars will appear hidden behind the shorter bars.

Example:



Due to limited space, the chart occasionally is printed in HIDDEN format even though the ORGANIZE value is CLUSTERED.

When the maximum value of a bar exceeds the value specified by YMAX, a ^ character will be printed on the tip of the bar. Similarly, a V character will be printed on the horizontal axis at the bottom of the bar when the minimum value of a bar is less than the YMIN value.

The location of each bar or cluster depends on the number of specified INPUT files. When only one file is specified, the bars or clusters will be distributed evenly on the horizontal axis. If multiple input files are specified, the bars or clusters will be arranged in increasing order according to the number of users identified in each input file. The distance between the bars or clusters will be proportional to the number of users that the bars or clusters represent. In cases where it is impossible to place all the clusters proportionally, Draw will make adjustments and issue a warning message.

5.3.3 Data Table

The data table is located below the figure and lists the actual value of each bar drawn on the chart. When one INPUT file is specified, an ID will be assigned to each row of the table to correlate the contents of the table with the figure. When multiple INPUT files are specified, each row will be identified by the number of users in each file. When there are more than 14 entries, the data table will print on a separate page.

5.3.4 Messages

Messages may appear below the table and include comments from the specification and any warning messages generated by Draw while processing the chart.

When the data table prints on a second page, messages will appear below the figure with an additional message indicating that the data table is on a separate page.

5.4 Draw Syntax

Now that we have an idea of the output that draw can create, let's look at the complete command syntax for Draw. The syntax is:

```
$ draw -
EMPOWER V3.1.8, Serial#R00000-000, Copyright PERFORMIX, Inc. 1988-95
Usage:
        draw  [-q] [-s specfile] [-o outputfile]
Options:
        -q    Uses Quick Interactive Mode
        -s    Uses batch mode with specfile as input specification file
        -o    Assigns outputfile as the output file
Notes:
        With no option specified, DRAW will enter the interactive mode.
            Interactive  mode  requires  *.STD  files  in  current  working
directory.
        The -o option will be ignored in interactive mode.
        Without -o option, the default output file is draw.out.
        The -q option can not be used together with the -s option.
Examples:
        draw
        draw -q
        draw -s sample.spec
        draw -s sample.spec -o sample.out
```

5.4.1 Default Interactive Mode

Interactive mode is the most common way to operate Draw. To enter this mode, specify the command `draw` without any options. You will be able to create specifications and generate bar charts on-line.

In interactive mode, Draw scans your current directory for a list of .STD files, then scans the files for performance measurements they contain. Draw uses this information to guide you through a series of questions that help Draw to produce a chart.

Note: Since Draw scans only your current directory, you must run Draw in the directory where your standard report files are stored.

When running Draw, you will be asked to select from a list of choices for each field. Since the list of choices is prepared from information in the standard report files, a default answer will be available for each question. You can accept a default value by pressing the RETURN key at each prompt.

As the interaction continues, each of your choices are verified. When all fields of a specification are obtained, the specification will be displayed on screen for verification. You can either save or discard the specification.

When the first specification is created and saved, you will be asked to enter the specification file name. The default is "draw.spec". If you choose to create additional specifications, the process will be repeated. All specifications created in the same session will be stored in the same specification file.

When you have completed all specifications, Draw will ask if your bar chart(s) should be generated. If you answer "no," Draw will exit from interactive mode, leaving you with just the specification file. If you answer "yes," you will be asked for the name of an output file; the default is "draw.out". Draw will generate a bar chart for each specification set in the specification file and exit.

5.4.2 Quick Interactive Mode

You may discover that the default values for the option statements, `COMMENT`, `LEGEND`, `ORGANIZE`, `TITLE`, `XTITLE`, `YMIN`, `YMAX` and `YTITLE`, are sufficient to execute Draw for your emulation. The `-q` option of the `draw` command specifies that these default values should be used. The resulting Draw execution can be completed more quickly than the normal interactive execution, since Draw will prompt you only for the required information.

Draw creates a specification file with the default name `draw.spec`. The output is placed in a file with the default name `draw.out`.

The following example illustrates using the `-q` option:

```
$ draw -q
Draw: EMPOWER V3.1.8, Serial#R00000-000, Copyright PERFORMIX, Inc. 1988-95

Please respond after each => prompt.
Press RETURN to accept a default.
Type quit at any time to exit.

Please select one or more INPUT file names.
Available files in the current directory are:
  1) 16user.STD   2) 1user.STD    3) 2user.STD
  4) 4user.STD    5) 8user.STD
Enter * to select all the files.
=> 1
Please select an EVENT to be charted.
  1) Scenario           2) Function           3) Transaction
Default is 1) Scenario
=> 3
Please select one or more Transactions for the X axis.
Available X axis values are:
  1) Overall     2) Unspecified    3) query     4)    update
Default is 1) Overall
=> 3 4
Please select one or more statistics for the Y axis.
Available Y axis values are:
  1) Total      2) Finish     3) Thruput    4) Median
  5) Average    6) Minimum    7) Maximum    8) Std-Dev
Default is 5) Average
=> 5 6 7
1 bar chart written to draw.out
Draw: normal completion.
$
```

5.4.3 Batch Mode with Specification File

The -s option of the draw command is used to identify the specification file for bar charts to be produced. The parameter of the -s option is the name of the specification file.

Typically, specifications are created in interactive mode, then batch mode is run to create the chart. If you want to create the same chart for several emulations, you should answer questions in interactive mode, edit and duplicate the specification with a system editor, then run Draw in batch mode to create charts for all emulations.

If multiple specifications are stored in your specification file, Draw will generate multiple bar charts separated by form feed characters.

5.4.4 Batch Mode with Output File Mode

The -o option identifies the output file for storing the generated bar charts. This option works only in batch mode (with the -s option) and is ignored in interactive mode. If no output file is specified, output is sent to the default file draw.out.

5.5 Specification Syntax

Specifications for bar charts may be created interactively by Draw (the most common method) or manually with any system editor. System editors often are used to copy and make minor changes to existing specifications.

The first statement of each specification read by Draw is BEGIN and the last statement is END. Multiple specifications may be stored in one file. Comments (not to be confused with the COMMENT statement) must begin with a pound sign "#" and may continue until the end of the line. Blank lines are allowed and are ignored during processing. Each BEGIN-END pair requires four statements that must appear on separate lines (see list below) and several optional statements.

Except in the case of input files, Draw does not distinguish between upper and lower case letters. The names of input files in the INPUT statement must be written exactly as they appear in your directory.

The required statements are:

O	INPUT	-	input files (the standard reports)
O	EVENT	-	event to be charted (scenario, function, or transaction)
O	X	-	list of scenarios, functions, or transactions
O	Y	-	performance measurements (average, maximum, P90, ...)

Each statement begins with a key word (statement name) and must be followed by a space and an equal sign:

Incorrect	Correct
`Event= Transaction`	`Event = Transaction`
`Event=Transaction`	`Event =Transaction`
`Event Transaction`	`EVENT =TRANSACTION`
`evENT = tranSAction`	

Note: If you operate Draw interactively, Draw will ensure that your specification has no mistakes.

5.5.1 INPUT

INPUT specifies the standard report files that Draw will read. When one file is identified, up to 60 X-axis values can be specified. The Draw tool will generate a bar chart that compares different events in the same input file.

If multiple input files are specified, only one X-axis value is allowed. In this case, the bar chart compares an event across several emulations and up to 60 input files may be specified.

Examples:

```
INPUT = 16user.STD
INPUT = mix25.STD, mix50.STD
```

5.5.2 EVENT

EVENT specifies the event to be charted. The value of EVENT can be Scenario, Function, or Transaction and only one value can be defined for each specification.

Examples:

```
EVENT = Scenario
EVENT = Function
EVENT = Transaction
```

5.5.3 X

X specifies a list of events that will appear on the X-axis of the bar chart. The names in the X value must appear in the first column of all INPUT files under the selected heading in the EVENT statement. If multiple files are specified in the INPUT statement, the X value must contain only one item. If the INPUT statement identifies only one file, up to 60 items can be specified .

Examples:

```
X = query, update
X = Overall
```

5.5.4 Y

Y specifies column headings of performance measures in the INPUT files. The selected measures will be extracted from the input files and plotted along the Y-axis. Up to three headings can be specified. The names must appear in the column headings of all input files in the same specified form.

Examples:

```
Y = Thruput
Y = Minimum, Average, Maximum
```

5.5.5 TITLE

TITLE specifies the title of the bar chart. The default is EMPOWER Bar Chart.

Example:

```
TITLE = Minimum and Maximum Response Time
```

5.5.6 XTITLE

XTITLE specifies the horizontal axis label. The default is the EVENT value.

Example:

```
XTITLE = Number of Emulated Users
```

5.5.7 YTITLE

YTITLE specifies the vertical axis label. The default is determined by the Y value as one of the following:

- Response time unit: as used in the input files. This value is chosen as the default when the specified Y value relates to response time (Median, Average, Std-Dev, P90, etc.).
- Count: is chosen as the default when the specified Y value relates to count (Total, Finish, W30, W50, etc.).
- Event/unit: where event is specified in the EVENT statement and unit is the time unit used in the input files. This format is chosen as the default when the specified Y value is throughput (Thruput).

Example:

```
YTITLE = Transactions/second
```

5.5.8 YMIN

YMIN specifies the minimum value of the vertical axis. If this statement is not specified, Draw will assign a value so that all bars on the chart are not lower than the horizontal axis. The actual chart value may not be the same as the specified value. Draw may adjust your value to improve the chart's appearance.

Example:

```
YMIN = 0.000000
```

5.5.9 YMAX

YMAX specifies the maximum value of the vertical axis. If this statement is not specified, Draw will assign a value so that all bars on the chart are not taller than the vertical axis. The actual chart value may not be the same as the specified value —Draw may adjust your value to improve the chart's appearance.

Example:

```
YMAX =10000000
```

5.5.10 LEGEND

LEGEND identifies each Y value with a single character and each bar will contain the corresponding LEGEND character. Performance measures have pre-defined default LEGEND characters.

Example:

```
LEGEND = *, !, &
```

5.5.11 ORGANIZE

ORGANIZE describes the format of bars to be drawn on the chart. The value of ORGANIZE can be CLUSTERED or HIDDEN. In the CLUSTERED format, all bars corresponding to the same event are grouped together with different legend characters for each bar. With the HIDDEN format, all bars that correspond to the same event are combined into a single bar with different legend characters (the taller bars will hide behind the shorter bars).

Occasionally, due to limited space, two clusters in a CLUSTERED chart will collide with each other. In this case, Draw will force the output to print in the HIDDEN format with a warning message.

The default value of ORGANIZE is CLUSTERED .

Example:

```
ORGANIZE = HIDDEN
```

5.5.12 COMMENT

COMMENT specifies a comment to be printed below the bar chart. It is the only statement that can be defined more than once in a specification. Comments are printed on the chart in the order they are written in the specification file. The maximum number of COMMENT statements is 50. An example follows:

```
COMMENT = This chart created by DL
```

6.0 Monitor

The Monitor tool allows you to monitor script activity during script execution. This tool is useful during script development and multi-user load tests because it interprets and displays significant status information for executing scripts.

Monitoring script activity during load testing is very difficult without Monitor because you must activate the display option for each executing script or browse through script logs to discover problems. The display option requires a terminal or workstation for each displayed script which would be extremely expensive, and browsing through log files is very time-consuming and inefficient.

Evaluating status information during a multi-user emulation is also very difficult without Monitor because you must rely on reports generated after the test. If resulting performance information is unsatisfactory, you would have to search through script log files to determine the cause of the problem. With Monitor, you can easily follow each script's activity and identify problems as they occur.

Benefits of using Monitor are:

- Simplified script debugging. Since you can monitor scripts as they execute, you can see how scripts have progressed. If a script becomes blocked, you immediately can detect the block, examine the log file of the script, or edit the source code. You do not have to exit Monitor to initiate edits.
- You can perform real time presentations during a load test.
- You can monitor response time on line as the scripts run.
- Script execution can be controlled dynamically.

6.1 Shared Memory

When your scripts are compiled with the Scc, Xscc, or Cscc tools (which automatically include the Monitor library functions), they will continuously update current status information in an area of UNIX shared memory. Monitor then displays the information in a useful format. More than one Monitor session can run simultaneously, and users can monitor their own scripts or all scripts running on the RTE machine.

The maximum number of scripts allowed in shared memory depends on the maximum size of a shared memory segment configured in your UNIX kernel (`SHMMAX`), and on the number given as an argument in the command to start Monitor. (A description of this argument is given in section 6.3.11.) The default size for the number of shared memory segments is 1024, with 4096 being the maximum.

Traditionally, script IDs are unique within a load testing session. Monitor enforces this uniqueness by allowing a script to "steal" the memory slot of another script having the same script ID. Therefore, if you run a test again after the first test completed, the second run will "re-use" the previous test's memory slot. Users must be careful to designate different script IDs. If two scripts with the same script ID are executed concurrently, the second script will overwrite the first.

6.2 The Curses Library

Monitor is written using the UNIX curses library. Therefore, any terminal that can run other curses applications (such as "vi") can run Monitor. The number of scripts that can be displayed on one screen is determined by the number of lines specified in your terminal's curses definition or the value of your `LINES` shell environment variable (if it is supported by your curses library implementation). To use Monitor, you must inform the system of your terminal type by setting your `TERM` variable (`term` for `csh` users).

For Bourne shell (sh) users, you can set your TERM variable by typing at the sh prompt:

```
TERM=vt100; export TERM
```

C-shell users would type at the csh prompt:

```
set term=vt100
```

or

```
setenv term vt100
```

Note: Of course, you would insert your appropriate terminal type in place of vt100.

Some curses implementations do not gracefully fail if you do not specify a valid terminal type before executing a curses application. Some of them simply fail to execute the application and return you to the UNIX shell with no indication that a problem occurred. Others may create more serious problems, such as dumping core. If a core dump occurs, you should verify that your terminal type is in the system's terminfo data base for System V or /etc/termcap for BSD UNIX. User-friendly curses implementations normally will print a message like the following if they detect that your terminal type is unknown or invalid:

```
<terminal type>: Unknown terminal type
```

or

```
I don't know enough about your <terminal type> terminal.
```

If Monitor does not recognize your terminal, the following message will appear:

```
I don't know enough about your terminal to run mon!
```

6.3 Monitor Syntax

This section describes usage information displayed from the UNIX shell. The usage menus for Monitor are displayed when you enter the command `mon` for EMPOWER, `xmon` for EMPOWER/X, or `csmon` for EMPOWER/CS, with a hyphen(-) as a parameter. The usage menus for each tool are identical.

```
$ mon -
EMPOWER V3.1.8, Serial#R00000-000, Copyright PERFORMIX, Inc. 1988-95
Usage:
       mon [-.ekorwAS] [-v n] [-i n] [-s n]
Options:
       -.            enter in dot mode
       -e            do not display exited scripts
       -k            kill processes attached to shared memory segment
       -o            do not display scripts owned by others
       -r            remove shared memory segment
       -w            wrap log before edit
       -A            use alternate character set (if available)
       -S            print status of shared memory segment
       -v n          enter onto view n
       -i n          enter with interval n
       -s n          create shared memory segment for n scripts
```

6.3.1 -.

The `-.` option starts Monitor automatically performing a '.' command. This command substitutes a single '.' character for control characters normally displayed as a two character (^c) sequence. This option makes the screen easier to read and allows more data to be displayed when it contains many control characters.

6.3.2 -e

The `-e` option specifies that exited scripts will not be shown in Monitor views during the session. The default mode displays all scripts. The `-e` option only displays executing scripts which allows you to focus on scripts that are still running without sorting through "dead" scripts or deleting scripts from shared memory.

6.3.3 -k

The `-k` option allows you to kill all currently running scripts. After entering this option, a verification statement will appear. You can kill only those scripts belonging to you. See the following example:

```
$ mon  -k

This procedure kills processes owned by you that are attached to the
EMPOWER shared memory segment.  Continue?
```

If you answer `y` to the prompt while scripts belonging to another user are running, the following message will be displayed:

```
The following processes can't be killed because you're not the
owner.

PID    Owner
1674   wendy
1681   wendy
2053   wendy
....
```

If you want to force the kill of all scripts (even those not owned by you), you can log in as the super user and type "`mon -k`".

6.3.4 -o

The `-o` option will start Monitor without displaying scripts that you do not own.

6.3.5 -r

The `-r` option allows you to remove the shared memory segment. This option will not start Monitor.

To remove the shared memory segment, you must own it (own either the first script to begin execution, or be the user who executes the `mon`, `xmon`, or `csmon` command, whichever occurs first), or be the super-user. If the shared memory segment is removed successfully, a message similar to the following will be displayed:

```
Shared memory segment ID 0 removed.
Shared memory semaphore ID 0 removed.
```

If scripts are running, you must use the `-k` option to kill all scripts before the shared memory segment can be removed.

If you are not the owner and you try to remove the shared memory segment with the `-r` option, the following message will be displayed:

```
The EMPOWER shared memory segment must be removed by its creator
<creatorname>.
```

or

```
The EMPOWER/X shared memory segment must be removed by its creator
<creatorname>.
```

You can force removal of the shared memory segment by logging in as the super-user and typing "`mon -r`".

If you are the owner of the shared memory segment, but scripts are still running and you try to remove the segment when scripts are still executing, the following message will be displayed:

```
There are 10 processes attached to the EMPOWER shared memory
segment.
```

All EMPOWER script and Monitor processes must be killed before proceeding.

6.3.6 -w

The `-w` option causes log files to be wrapped before they are viewed.

6.3.7 -A

The `-A` option specifies that the alternate character set (line drawing characters) will be used if available on your system and for your terminal. Using the line drawing characters allows such features as a box drawn around the help windows.

This function may not work on your terminal correctly (or at all) in the following three cases:

- Although your terminal may have a line-drawing character set, your version of `curses` may not support it.
- Your terminal may have to be initialized into a mode that enables you to use the line-drawing character set. For instance, the vt100 requires that its alternate character set be assigned to the line-drawing character set by outputting the escape sequence `<ESC>)0` to the terminal. If you are not initialized into a line-drawing character set mode, you may see normal alphabetic characters in places where line-drawing characters should be, such as around the help screen.
- The `curses` terminal definition for your terminal may not be correct.

Because of the third reason listed above, specifications of -A are optional rather than the default.

6.3.8 -S

The -S option will print the status of the shared memory segment, as in the following example:

```
$ mon -S

EMPOWER V3.1.8 Serial#R00000-000 Copyright PERFORMIX, Inc. 1988-95
Maximum number of sessions available in this shared memory segment = 1024
Number of sessions currently attached to the shared memory segment = 1
Size of shared memory segment in bytes = <491440>
Owner of shared memory segment = <ownername>
```

The maximum number of sessions that can be monitored depends on your system. Normally, PERFORMIX sets the maximum at 4096 before the software is shipped and this amount cannot be changed.

The maximum number of sessions available is less than or equal to the maximum number of sessions that can be monitored. The default value is 1024.

The number of sessions currently attached to the shared memory segment is the number of scripts and Monitor sessions currently running.

The owner of the shared memory segment is the first user to execute the EMPOWER, EMPOWER/X, or EMPOWER/CS command that created the shared memory segment. This user will be either the first person to run a script or run Monitor.

6.3.9 -v

Every time you start Monitor, you will enter View 1 by default. The `-v n` option allows you to enter a specified view.

```
$ mon -v 3
```

6.3.10 -i

You can use the `-i n` option to start Monitor with the time interval n. Information on the current screen will be updated every n seconds. The default interval value is five seconds, the maximum interval value is 3600 seconds, and the minimum value is one. If you specify an interval value exceeding the maximum, the interval will be set to the maximum value. If you specify an interval value of zero, the interval will be set to the default.

```
$ mon -i 2
```

6.3.11 -s

The `-s n` option allows you to initialize the number of shared memory segments. The minimum value of n is four and the maximum value normally is set at 1024. This option is useful if you do not have enough shared memory available in your UNIX kernel or if you want to limit the number of scripts that can be monitored to display a sample of the running scripts. Scripts started when the shared memory segment is full will not be monitored.

Example:

```
$ mon -s 64
```

Increasing the maximum number of sessions available in the shared memory segment is also useful if you are running an extremely large test.

```
$ mon -s 2048
```

6.4 Starting Monitor

After a script is compiled, you use the `mix` command to begin multi-user testing. While the test is running, you can use Monitor to monitor script execution. Because Monitor is an interactive program, you can control script execution by entering various Monitor commands.

The command to start Monitor for EMPOWER is `mon`; for EMPOWER/X, `xmon`; for EMPOWER/CS, `csmon`.

If you see the following response:

```
mon:  not found
```

then your UNIX shell `PATH` variable (`path` for `csh` users) does not include the directory where Monitor has been installed.

If the correct directory is `/usr/empower`, you would enter:

For Bourne Shell users:

```
$ PATH=/usr/empower/bin:$PATH  export  PATH
```

For C Shell users:

```
$ set  path=(  $path  /usr/empower/bin  )
```

Monitor begins operating in an interactive view mode which displays status information in tabular format. When you begin a monitoring session, View 1 will appear on screen.

Example view modes for each monitoring tool follow:

For EMPOWER:

```
Mon Feb 11 10:43:16          EMPOWER MONITOR V3.1.8    (c) PERFORMIX, Inc. 1995
View 1 of 7  Script 1 of 8          Running      ScriptId Sorting ^  Interval 5

ScriptId __Script State ____________LastXmit ____________________________LastRcv
   user1| manager| xmit| country.^[kkA and w|[4hr country.^[[41^H^[[A^H^H^H^[[An
   user2|  typist| xmit|^H^H^H^Hof their cou|hto come to the aid ^[[41^M^[[B^[[K
   user3| manager|think|o come to the aid ^M|hto come to the aid ^[[41^M^[[B^[[K
   user4|  typist|  rcv|^Mfor all good men^M|   ^[[K^[[4hfor all good men^[[41^M
   user5|  typist|  rcv|^Mfor all good men^M|[[4hfor all good men^[[41^M^[[B^[[K
   user6| manager| xmit|^Mvi^MiNow is the ti|^H~^[[B^H~^[[B^H~^[[B^H~^[[B^H~^[[H
   user7| manager|  rcv|ry.^[kkA and women^[|6\344\240wo\355e\356^[\333\2641\210
   user8|  typist| xmit|.^[kkA and women^[:q|[A^H^H^H^[[An^[[4h and women^[[41^H
```

09567694.102600

For EMPOWER/X:

```
Fri Apr 16 16:41:51        EMPOWER/X MONITOR V4.1.4    (c) PERFORMIX, Inc. 1995
View  1 of 10  Script 2 of 4         Running      ScriptId Sorting ^  Interval 5

ScriptId __Script _____State Cl ______________LastXmit __________________LastRcv
      u1  script1  keystring  2 Mls^Mdate^Mwho^Mpwd^Mt  /Xstuff^J[empower@nomad]
      u2  script1    textrcv  2                ^Mls^M erycode.Jeff^Jquerycode.S
      u3  script1 disconnect  1             <LEFT_ALT>              PPPPPPPPPPPP
      u4  script1    textrcv  2      ^Mls^Mdate^Mwho^M  6 10:18^J[empower@nomad]
```

For EMPOWER/CS:

```
Wed Jan  4 15:33:43        EMPOWER/CS MONITOR V1.0     (c) PERFORMIX, Inc. 1995
View 1 of 7  Script 1 of 4          Running       ScriptId Sorting ^  Interval 5

ScriptId __Script State ____________________________LastEvent _____CurrentWindow
   user1       foo  type            <SysKeyPress>James Smith            Employee
   user2       foo event      <SysKeyPress><LeftButtonPress>                 Run
   user3       foo event                       <SysKeyPress>                Help
   user4       foo  type           <SysKeyPress>test.dat^M               Save As
```

6.5 Monitor Help Screens

You will begin your Monitor session in view mode. From this mode you can type '?' to display the help menu listing Monitor commands. The help menu is displayed in three help screens. You can press the space bar to toggle between the help screens or press a 'q' to exit the help screen. The three help screens for the EMPOWER Monitor are shown below.

(*Note:* Some of these commands may not appear on the Monitor help screens for EMPOWER/X and EMPOWER/CS which only list the commands particular to those products.)

```
Moving______________     Scrolling____________    Searching_____________
  h  left                  ^u half screen up        /   search
  j  down                  ^d half screen down      \   reverse search
  +  down                  ^f screen forward        n   next search
  k  up                    ^b screen backward       |   filter out scripts
  -  up                    ^e one script down     Other_________________
  l  right                 ^y one script up         ^[  forget input
  nv to view n           Escaping_____________      ^l  refresh
  nG to script n           !  shell                 ni  set interval to n
  G  to last script        T  trace script          s   change sort field
  H  highest on screen     E  edit script           C   screen capture
  M  middle of screen      V  view log              d   enable display
  L  last on screen        Z  zoom to script        q   quit
                                                    ?   help
  Press Spacebar to see next help, q to quit help...
```

```
Deleting________________ Killing_________________ Flushing_________________
 nDD delete n scripts     nKK kill n scripts       nFF flush n logs
   Ds delete screen         Ks kill screen           Fs flush screen
   De delete exited         Ka kill all              Fa flush all

Resuming________________ Suspending______________ Interrupting_____________
 nRR resume n scripts     nSS suspend n scripts    nII interrupt n scripts
   Rs resume screen         Ss suspend screen        Is interrupt screen
   Ra resume all            Sa suspend all           Ia interrupt all
   nRt set TRESUME to n                              nIt set TINTR to n
                         Joining_________________
                            JD join on used display
                            JJ join on this display
                            ~. complete join
    Press Spacebar to see next help, q to quit help...
```

```
                                Toggles_______________
                                .  dots
                                a  anchor mode
                                b  bars
                                c  compressed
                                e  exit mode
                                o  owner mode
                                p  pause
                                r  relative time
                                w  wrap log before view
                                x  sort order
    Press Spacebar to see first help, q to quit help...
```

The following sections describe commands you can use during Monitor sessions. Many commands accept optional numeric arguments preceding them. Those commands are displayed on the help screen with 'n' preceding the command character, where 'n' stands for the optional number.

6.6 First Help Screen Commands

6.6.1 Moving

In view mode, moving commands function similarly to "vi" editor commands.

`j, +`	Move the cursor n scripts down
`k, -`	Move the cursor n scripts up
`h`	Change the view to one view to the left wrapping at the end (If 1 is the current view, `h` will change the view to View 7.)
`l`	Change the view to one view to the right wrapping at the end (If 7 is the current view, `l` will change the view to View 1.)
`nv`	Change the view to view `n` (If the current view is 2, `4v` will change the view to view 4. The default value for `n` is current view + 1.)
`nG`	Move the cursor to the `n`th script scrolling as necessary (The default value for `n` is last.)
`H`	Move the cursor to the highest script on the current screen
`M`	Move the cursor to the middle script on the current screen
`L`	Move the cursor to the last script on the current screen

6.6.2 Scrolling

The scrolling commands operate similarly to "vi" editor commands:

`^u`	Move up a half screen
`^d`	Move down a half screen
`^f`	Move forward a full screen
`^b`	Move backward a full screen
`^e`	Move one script down
`^y`	Move one script up

Note: You can not scroll the screen when using the 'p' command which pauses Monitor.

6.6.3 Searching

The searching commands search for specified data within the current sort field. If you enter an incorrect search key, you can use the backspace key to remove it. For this command, 'n' is the letter n and does not designate a number.

/	Forward search
\	Backward search
n	Next search (Search for the next item in the same direction as the previous search.)

If you try to search when paused, the following message will appear:

```
Can't search when paused
```

If the searched-for data cannot be found (make sure you are searching for data in the current sort field), the following message is displayed:

```
Not found below in <scriptid> column
```

If you specify next search but have not performed a previous search, the following message will appear:

```
No previous search pattern
```

6.6.4 Escape Modes

The escape commands allow you to enter modes external to the view mode without quitting your Monitor session. After exiting an escape mode, you will return to view mode. The `shell` escape mode allows you to escape temporarily to a UNIX shell to execute shell commands; `trace` allows you to trace a script source code while the script executes; `edit` allows you to edit the script source code at the current execution location; `view` allows you to flush the script's log file buffer and view the log file; and `zoom`, only available for `mon`, displays the current screen of a script.

The escape commands are:

!	shell escape mode command
T	trace escape mode command
E	edit escape mode command
V	view escape mode command
Z	zoom escape mode command (only available for mon)

The zoom escape mode works only if the following function is in your script source file:

```
Term(ZOOM, VT100|LINES24|AUTOWRAP)
```

You can enter the shell escape mode at any time. To enter other modes, you must move the cursor to the required script and press the escape command.

6.6.4.1 Trace Mode

When a script has been idle for an unusual amount of time, tracing its execution to see where the script is "blocked" can be very useful. The "T" command allows you to see what line in the script source file the script currently is executing.

Tracing a script assumes that you have used the Beginsource() and Endsource() functions properly. Beginsource() should be the first function executed whenever a script branches to a new source file. Endsource() should be executed just before a return. Beginsource() requires an argument - the name of the source file without the .c extension. Endsource() has no argument. Source files may be nested.

While tracing a script, you can use the "?" command to get the trace mode help menu:

```
    _Trace Help_

p  toggle pause
ni set interval to n
^l refresh
l  toggle line numbers
?  help
q  quit
Press q to quit help...
```

Six commands are available in trace mode:

`p`	Pauses the cursor (Tracing will stop temporarily until the `p` command is pressed again.)
`ni`	Sets the update interval to `n` seconds
`^l`	Refreshes the trace screen
`l`	Displays the source script's line numbers on the trace screen (Pressing the `l` command again will turn the line number display off.)
`?`	Displays the help menu
`q`	Quits the Trace mode or quits the Help menu

6.6.4.2 Edit Mode

The 'E' command allows you to invoke the "vi" editor on the script source file. The "vi" editor must be located either in `/usr/bin/vi`, or your `UNIX $PATH` (`$path` for `csh` users) must include the location of "`vi`".

When entering the edit mode, the cursor will be placed on the line currently executing. Once you have entered the edit mode, you are in the vi editor's command mode. You must know how to operate the "vi" editor before using the edit mode.

Note: Any changes made to the script will be included only after you recompile the script with the Cscc command and then execute it.

6.6.4.3 View Mode

When a script has been idle for a long duration, looking at the script's log file is useful. The `v` command requests that the script flush its log file buffer before invoking the UNIX `view` command on the log file. Since the `v` command will cause the script to flush its buffer, it may not be necessary for you to specify the `Set(NOBUF)` function in your script. This will increase performance during multi-user tests since `Set(NOBUF)` will cause the script to flush the buffer frequently.

6.6.4.4 Zoom Mode

The `z` command can display the screen image of an emulated user (Only available for the Monitor for EMPOWER). The screen image is a snapshot of what a user would see if he or she were running the script in display mode at a terminal.

Before the zoom can occur, the user being "zoomed" must have been updating a copy of the screen initiated with the `Term()` function in a script. The first argument to `Term()` must be `ZOOM`, and the second must describe the terminal that the system under test (SUT) thinks it is updating, for example, `Term(ZOOM, VT100|LINES24|AUTOWRAP)`. The script will interpret the escape sequences sent to it and maintain a copy of the user's screen.

Valid terminal types for the `Term()` function are `VT100`, `VT220`, `WYSE50`, `WYSE60`, and `ANSI`.

Frequency of the screen image is determined by the `zoom` interval, which is every five seconds by default. This interval can be changed by using the "`i`" command within the `zoom` mode.

```
    _Zoom Help_

p  toggle pause
ni set interval to n
^l refresh
?  help
q  quit
```

Five commands are available in the EMPOWER Monitor's `zoom` mode:

`p`	Pauses the screen image (The screen update will stop temporarily until the `p` command is pressed again.)
`ni`	Sets the update interval to `n` seconds
`^l`	Refreshes the `zoom` screen
`?`	Displays the help menu
`q`	Quits zoom mode or quits the help menu

6.6.5 Other Commands

6.6.5.1 ^[

`^[` (Esc) can be used to ignore input such as a typed number that you wish to discard.

6.6.5.2 ^l

`^l` (control-L) refreshes the screen.

6.6.5.3 ni

`ni` sets the display interval to `n` seconds. The display will be repainted with current information every `n` seconds. The default interval value is five seconds, the maximum interval value is 3,600 seconds, and the minimum is one. If you specify an interval value exceeding the maximum, the interval will be set to the maximum value. If you specify an interval value less than the minimum, the interval will be set to the default. When sorting by a field that may change often, we suggest that you do not set the time interval to a small value (such as 1i, 2i, or 3i).

6.6.5.4 s

s allows you to change the sort field (located on the second line of the header in Monitor) to a field on the current view. The default sort field is ScriptId where all the scripts will be listed by ScriptId. You can use the h and l keys to cycle through available fields. The ^[(Esc) key can be used to abort changing the sort field. Once you have selected the right field, press the Enter key to re-sort. Monitor pauses while the sort field is being selected.

When sorting on a field other than ScriptId, the ScriptId is used as a secondary sort field. Scripts will be sorted by the primary sort field first, and any scripts in which the primary sort field values are equal will be sorted again by the ScriptId. An exception to this process occurs when sorting by idle time; scripts that are in the "exit" state or the "suspend" state are sorted to the bottom since idle time is not applicable for these scripts. These scripts are sorted by ScriptID at the bottom of the list (i.e., after all scripts in other states).

To watch those scripts that are not progressing as anticipated, sorting by idle time is common. Scripts in a Rcv state with considerable amounts of idle time may be encountering problems on the SUT that need to be fixed. If your script is in a think state listed under the State column, idle time is expected.

Monitor displays a "snapshot" of what occurs in executing scripts, but script data changes constantly. For this reason, scripts may appear out-of-sort if data changes between the times data are sorted and painted on the screen. Also, time values are displayed as HH:MM:SS.hh where hh indicates hundredths of a second. When sorting by time values, and the times grow so that hundredths of a second no longer display, scripts may seem out of sort because the time values appear the same but scriptids are not in order. Although the hundredths value is not displayed, it continues to be used for calculating the sort order.

If a script runs for more than 24 hours, sorting by start time may seem out of order because the date portion of the time is not displayed.

Whenever many scripts appear to have completed execution, you may want to sort by `State` or `StatePattern` to help you isolate scripts that have terminated abnormally. Sorting by either of these fields while scripts execute is meaningless (and somewhat annoying) because data changes so often that it is obsolete as soon as it displays.

6.6.5.5 C

`c` allows you to capture the current screen into a specified file. You will be prompted with the file name the first time you issue the `c` command in the current session. The next time you invoke the `c` command, the current screen will append to the specified file.

6.6.5.6 d

`d` enables the display option for a script in the Monitor for EMPOWER. You can specify a port to watch the script execute on a separate terminal. The display option operates differently for the EMPOWER/X and EMPOWER/CS products. Refer to Sections X for details on operating the EMPOWER/X and EMPOWER/CS Monitor displays.

The display tty is included in View 4 under the Display field.

To initiate the display mode, move the cursor to a script in Monitor and enter the `d` command. Then, enter the name of a display TTY. You can specify a real TTY, for example `/dev/tty04`, or a pseudo TTY, for example `/dev/ttyp1`, where `/dev` is optional. The base name of the TTY used for display mode can be found in Monitor View 4.

To terminate display mode, enter the `d` command and press return, but without entering a name when prompted for a TTY.

You should display to a terminal of the same type used by the emulated user. For instance, if the SUT is sending escape sequences to the script destined for a VT100, you should display to a terminal that can interpret these sequences.

You can initiate display mode for a script that has been updating a copy of the screen itself, that is, it executed `Term(ZOOM, ...)`. When you initiate display mode for such a script, a copy of the screen will be sent to the TTY which will make the display as current as possible — you will not have to wait for the application on the SUT to refresh the screen.

The `Term()` function instructs EMPOWER to maintain a run-time screen image for use by any of the screen receive functions. This function also allows script execution to be viewed with the Zoom mode.

The syntax of `Term()` is:

```
Term(ZOOM, VT100|LINES24|AUTOWRAP);
```

The default `Term()` function placed in a script specifies `NOZOOM`. The other parameters depend on the terminal type used when creating the script and specify the initial state of the terminal. Valid parameters are `LINES24` or `LINES25` and either `AUTOWRAP` or `NOAUTOWRAP`, with `LINES24` and `AUTOWRAP` as the default.

6.6.5.7 q

`q` quits from any help menu or quits the Monitor session.

6.6.5.8 ?

? displays the help menu.

6.7 Second Help Screen Commands

6.7.1 Deleting

When scripts complete execution, you can use the delete commands to remove them from shared memory which allows you to focus only on those scripts that are still running. To delete a script, you must either be the script owner or super-user and the script must be in the "exit" state.

Syntax for the delete commands is listed below.

`nDD`	Delete `n` scripts starting with the current script (The default value for `n` is one.)
`Ds`	Delete all scripts on the current screen that belong to you
`De`	Delete all scripts in the "exit" state that belong to you

If you do not own the script you attempt to delete, the following message will appear on the third line of the header:

```
Not owner!
```

If you specified an `nDD` command where `n` is too large, you will see the following message:

```
Not that many scripts
```

If no scripts were found, the following message will appear:

```
Nothing to delete!
```

If you tried to delete a running script, the following error message is displayed:

```
Script not exited!
```

If you attempted a `Ds` or `De` command and no exited scripts were found, the following message will be displayed:

```
No exited scripts!
```

6.7.2 Killing

The kill commands can be used to "kill" currently executing scripts. To operate this command, the script must be executing (not exited), and you must be either the owner of the script or the super-user. The kill command sends a signal to the running script and the script will enter the "exit" state.

`nKK`	Kill n running scripts starting with the current script (The default value for n is one.)
`Ks`	Kill all running scripts on the current screen that belong to you
`Ka`	Kill all running scripts that belong to you

If you specify the kill command but do not own the script, the following warning message will be displayed:

```
Not owner!
```

If you are the owner but the script is in the "exit" state, `KK` will display the following message:

```
Script is already dead!
```

If no scripts are running, `Ks` or `Ka` will display the following message:

```
No live scripts!
```

6.7.3 Flushing

By default, each script performs buffered writes to its log file to increase performance during multi-user tests. The flush commands can be used to signal a script to flush its log file buffer which is useful if you are browsing through logs and need to see the most recent information in the log. The V command requests that the script flush its log file automatically before invoking the UNIX view command.

The syntax for the flush commands is listed below.

`nFF`	Flush n running scripts starting with the current script (The default value for n is one.)
`Fs`	Flush all running scripts on the current screen that belong to you
`Fa`	Flush all running scripts that belong to you

6.7.4 Resuming

Once a script is suspended, you can resume it using commands from Monitor or Mix. When resuming more than one script from Monitor, the scripts will resume in the background at intervals specified by the `nRt` command. When resuming from the Mix prompt, the scripts will resume according to the time specified in the Mix variable `tstart` (See Mix Section 2.x for more information).

`nRR`	Resume n suspended scripts (The default value for n is one.)
`Rs`	Resume all suspended scripts in the current screen that belong to you
`Ra`	Resume all suspended scripts that belong to you
`nRt`	Set the resume interval to n seconds (The default value for n is five.)

6.7.5 Suspending

You may suspend a script while it executes if you are the owner of that script. Once a script is suspended, it will remain suspended until it is resumed.

`nSS`	Suspend n scripts (The default value for n is one.)
`Ss`	Suspend all scripts on current screen that belong to you
`Sa`	Suspend all scripts that belong to you

6.7.6 Interrupting

Your script may become blocked in a receive state during your EMPOWER Monitor session. If so, you can interrupt the script with the interrupt commands. Interrupting will force a timeout in the script, and the script will continue if possible. If you have set a large timeout value, interrupting is useful if you do not want to wait for the timeout interval to expire before executing subsequent script transactions.

The syntax for the interrupt commands is listed below.

`nII`	Interrupt `n` blocked scripts starting with the current script (The default value for `n` is one.)
`Is`	Interrupt all blocked scripts on the current screen that belong to you
`Ia`	Interrupt all blocked scripts that belong to you
`nIt`	Set the interrupt interval (The default value for `n` is zero.)

If you interrupt a script that is not blocked, you will receive the following message:

```
Script is not in a rcv state!
```

6.7.7 Joining

The Join feature of Monitor (which applies only to the Monitor for EMPOWER) allows you to become the emulated user of a blocked script. You simply select a script and your display will become that of the emulated user. You can interact with the application on the SUT in an attempt to revive it, determine what went wrong, or simply put the application back on track so the script can continue. With the Join feature, you will not have to abort tests simply because one or two scripts fail, and developers who need to resolve problems that occur only under load can debug them while a test continues. All keystrokes entered during the join are recorded so you can incorporate them for subsequent script runs.

A script can be joined in three ways: Two ways are from Monitor and are performed dynamically; one way is from within the script and occurs at a predetermined location. These methods are discussed in more detail in the following sections. Before a join from Monitor can occur, a script must be blocked in a receive state or be suspended. You can use the `ss` command from `mon` to force a script to suspend if it does not reach a blocked state voluntarily.

6.7.7.1 Joining from Monitor

To begin a joined session in Monitor when the script is in display mode (initiated with the `-d` or `-D` option of the script, or with the `d` command of `mon`), you can move the cursor to the script and type `Jd` (which stands for join display). The joined session will begin on the real or pseudo display terminal. You now can move to the terminal and begin typing.

A script will pause until you end a joined session by typing "`~.`" at the terminal. The script then will return to its previously blocked state. If the script was suspended, you can resume it with the `RR` command of `mon`. If the script was blocked, you can force a timeout with the `II` command of `mon`, thereby completing a `Rcv()`. An interrupted script will not exit with `II`, even if the timeout condition was set to `EXIT`, but will continue as though the timeout condition was set to `CONTINUE`. You must enter the `KK` command to force a script to exit.

If a script is not in display mode, you can begin a joined session in Monitor with the `JJ` command. In this case, the Monitor display will become the emulated user's display. The Monitor display will clear, and if the script was executed with a `Term(ZOOM, ...)` function, the emulated user's display will be presented to you. You then can begin typing, ending the joined session with "`~.`". If the script was not executed with `Term(ZOOM,...)`, the display will remain blank and you must enter a command that will force the application to refresh its display.

Note: You should join a script only from a terminal that is the same type set by the emulated user on the SUT. Output from the SUT and the function keys that you enter will work only when the terminal types are consistent.

6.7.7.2 Joining from within a Script

You can join a script in a predetermined location by including the `Join()` function in a script. When your script encounters the `Join() function`, a joined session will begin. Such a script must be executed in the foreground, preferably with the `-d` or `-D` option, because `Join()` will be ignored if the script runs in the background.

Keystrokes that you enter during a joined session are recorded in a `.x` file with the prefix "`mon.`" followed by the script's scriptid. The scriptid is taken from the Mix table, the `-s` option of the script, or the name of the script if it was run from the command line. If an `.x` file exists, the keystrokes will be appended to the file.

The keystrokes and responses that occurred during a joined session will be built into a portion of the corresponding script. The script's prefix will be "`mon.`" followed by the script ID with a `.c` extension. EMPOWER will attempt to group the keystrokes into logical transactions. If the `.c` file exists, the new portion of the script will be appended to the file. You can use the `.x` or `.c` files built during the joined session to aid in patching a script so that it can handle new behavior from an application on the SUT.

6.8 Third Help Screen Commands

The toggle commands affect overall operation of Monitor.

.	converts all two-character control characters to dots making the screen easier to read and allowing more data to fit on the screen.
a	anchors the script. The < sign is used to indicate an anchored script and is displayed in the column between the ScriptId and the Script name. When sorting on fields that often change, the anchor is useful for continuing to monitor a particular script.
b	adds a vertical line between fields to discriminate more easily between columns of data. If the screen will be repainted often, do not use this command because printing the column separator will slow painting the screen.
c	compresses all views of one script onto the screen. To choose a script to compress, move the cursor to that script and press 'c'. For `mon` and `csmon`, press 'c' again to toggle back to the original, uncompressed view of all scripts. For `xmon`, press 'c' a second time to remove blank lines. If your screen originally was not large enough to display information from all 10 `xmon` views, this process will allow you to view more of the current script's status

	information. Then, you can press 'c' a third time to toggle back to the original, uncompressed xmon view. The compress command is useful for monitoring all information of a single script. To look at the compressed view of another script, simply move the cursor to the appropriate script.
e	toggles exit mode. The default mode displays all scripts. The alternate mode displays only executing scripts. This command allows you to focus on scripts that are still running without having to sort through "dead" scripts or delete scripts from shared memory.
o	toggles owner mode. The default mode displays all user scripts. The alternate mode displays only scripts that belong to you.
p	pauses display, since script information changes often. You may move to another script when paused, but you cannot scroll the screen. The ? (help) command also will pause Monitor.
r	toggles between absolute and relative times. This is useful when displaying timestamps in view 5 of mon and csmon and view 6 of xmon.
w	toggles between wrapping and not wrapping logs before viewing. Wrapping is useful if your log includes responses from the SUT which are too long to be viewed by a system editor or which contain unprintable characters.
x	toggles between ascending (^) and descending (v) sort order.

6.9 Views

Seven different views are available for mon, ten are available for xmon, and seven for csmon. Each view displays a screen with a table of information for all scripts currently running or recently exited and all views display information for the same set of scripts. Each script requires one line on your screen. If you have more scripts than lines on your screen, you can scroll to the remaining scripts.

The first two lines of each view represent the header which includes the date, product information, view number, script number, current state, current sorting field, current sorting order, and the update interval. The third line displays messages for your information as you execute commands. The fourth line of each view contains headers for the different displayed fields .

When Monitor starts, it enters view mode in a "Loading" state until it has initialized to begin monitoring. Next, it will go into a "Waiting" state until it has scripts to monitor. As scripts begin to execute, it will go into a "Running" state. Other possible states are "Paused" and "Sorting". These states are displayed in view mode.

6.9.1 EMPOWER Monitor Views

The seven views available in the Monitor for EMPOWER (mon) are described in this section.

6.9.1.1 EMPOWER Monitor View 1

View 1 of `mon` displays the state of each script and the data transmitted and received by each script.

```
Mon Feb 11 10:43:16          EMPOWER MONITOR V3.1.8      (c) PERFORMIX, Inc. 1995
View 1 of 7  Script 1 of 8           Running      ScriptId Sorting ^  Interval 5

ScriptId __Script State ____________LastXmit _____________________________LastRcv
   user1| manager| xmit| country.^[kkA and w|[4hr country.^[[41^H^[[A^H^H^H^[[An
   user2|  typist| xmit|^H^H^H^Hof their cou|hto come to the aid ^[[41^M^[[B^[[K
   user3| manager|think|o come to the aid ^M|hto come to the aid ^[[41^M^[[B^[[K
   user4|  typist|  rcv|^Mfor all good men^M|   ^[[K^[[4hfor all good men^[[41^M
   user5|  typist|  rcv|^Mfor all good men^M|[[4hfor all good men^[[41^M^[[B^[[K
   user6| manager| xmit|^Mvi^MiNow is the ti|^H~^[[B^H~^[[B^H~^[[B^H~^[[B^H~^[[H
   user7| manager|  rcv|ry.^[kkA and women^[|6\344\240wo\355e\356^[\333\2641\210
   user8|  typist| xmit|.^[kkA and women^[:q|[A^H^H^H^[[An^[[4h and women^[[41^H
```

`ScriptId`	A unique identifier specified in the script table and used by the Mix tool (If you run a script from the command line, the `ScriptId` will be the same as the script name.)
`Script`	The name of the compiled script (The source version of the script typically is found in `script.c`.)
`State`	The current state of the script's execution (`xmit`, `rcv`, `suspend`, `think`, and `exit`)
`LastXmit`	The last characters transmitted to the SUT by the script (Characters scroll from right to left as they are transmitted so that the rightmost character is the last character transmitted.)
`LastRcv`	The last characters received from the SUT by the script (Characters scroll from right to left as they are received so that the rightmost character is the last character received.)

Monitor enforces that `ScriptId`'s are unique by allowing a script to "steal" the Monitor slot of another script having the same `ScriptId`. If you run one load test after the first is completed, the second run will "re-use" the first Monitor slots. Users must be careful to designate different script IDs.

Sorting by State is useful when most scripts are in the same state or entering the same state, as when scripts are exiting or suspending.

When many control characters are displayed, entering '.' (dot) mode will allow more characters to be displayed in the `LastXmit` and `LastRcv` fields.

6.7.1.2 EMPOWER Monitor View 2

View 2 of mon displays detailed information of script execution states, idle time, and the number of warning and timeout messages for each script. This view often is used for debugging scripts.

```
Mon Feb 11 10:47:37          EMPOWER MONITOR V3.1.8     (c) PERFORMIX, Inc. 1995
View 2 of 7  Script 1 of 8           Running           Idle Sorting v  Interval 5

ScriptId __Script State ___________StatePattern ____Idle NWarn NTo __ToCondition
   user5   typist   rcv                      XXX   13.97              15 continue
   user7  manager   rcv                        ]    2.27            1 15 continue
   user1  manager think                     1.83    1.33            1 15 continue
   user8   typist  xmit                   who^M      .58              15 continue
   user6  manager  xmit      for all good men^M      .56              15 continue
   user3  manager  xmit ^H^Hof their country.^[      .54            1 15 continue
   user2   typist think                     1.78     .51            1 15 continue
   user4  manager think                     1.33              1    4 15 continue
```

StatePattern	Additional information relative to the script execution state, such as the string being transmitted (Xmit), the string being looked for (Rcv), the current think value (Think), or the exit status (exit)
Idle	The time, in seconds, that the script has been idle or the time since the shared memory segment was last updated by the script
NWarn	The number of EMPOWER warnings issued to the script (early pattern matches, etc)
NTo	The number of EMPOWER timeouts issued to the script (i.e. when an expected rcv string never arrives)
ToCondition	The number of seconds before a timeout will occur and the action to be taken upon timeout

6.9.1.3 EMPOWER Monitor View 3

View 3 of `mon` displays script execution completion percentage, the script source file name, the log file name, the communication port used by the script, and any script notes.

```
Fri Apr 16 14:04:45          EMPOWER MONITOR V3.1.8      (c) PERFORMIX, Inc. 1995
View 3 of 7  Script 1 of 4          Running       ScriptId Sorting ^  Interval 5

ScriptId __Script Line NLine _Pct ______Source _________Log _______Port ____Note
   user1 example1   46    46 100.     example1        log1 t:localhost
   user2       ss   46    46 100.           ss        log2 t:localhost
   user3       db   46    46 100.           db        log3 t:localhost
   user4     comp   46    46 100.         comp        log4 t:localhost
```

`Line`	The line in the script source (.c) file currently executing
`NLine`	The number of lines in the source (.c) file
`Pct`	The percentage of script completion
`Source`	The name of the script source file (may be a function source file)
`Log`	The log file to which the script is writing
`Port`	The port on the RTE through which the script is communicating
`Note`	A user-defined note placed in the script by using the `Note()` function (A note can be placed at any location in the script and changed as appropriate.)

6.9.1.4 EMPOWER Monitor View 4

View 4 of mon displays general information about the scripts, such as the owner of the script, think time distribution, type rate, and the amount of I/O traffic information. I/O information can be used in workload analysis.

```
Fri Apr 16 14:05:20          EMPOWER MONITOR V3.1.8     (c) PERFORMIX, Inc. 1995
View 4 of 7  Script 1 of 4          Running        ScriptId Sorting ^  Interval 5

ScriptId __Script __Pid ___Owner NCXmit ___NCRcv _IO _Type _Disp ______ThinkTime
   user1 example1   499  empower     68     5441  80        ttyq2 uniform 10. 25.
   user2       ss   501  empower     67     4464  66              uniform 10. 25.
   user3       db   503  empower     66     2708  41              uniform 10. 25.
   user4     comp   506  empower     68     2765  40              uniform 10. 25.
```

Pid	The process ID of the script
Owner	The log in ID of the script's owner, (i.e., the person who started the script)
NCXmit	The total number of characters transmitted to the SUT
NCRcv	The total number of characters received from the SUT
IO	I/O ratio of the script or the number of characters received for every one character transmitted
Type	The typing speed of the script
Disp	The display terminal, if applicable
ThinkTime	The think time distribution and parameters

6.9.1.5 EMPOWER Monitor View 5

View 5 of mon displays times of various events as well as elapsed time information. Users often can estimate load test completion time by looking at this view.

```
Mon Feb 11 10:54:06         EMPOWER MONITOR V3.1.8     (c) PERFORMIX, Inc. 1995
View 5 of 7  Script 4 of 8          Running       ScriptId Sorting ^  Interval 1

ScriptId __Script ____Start _Suspend __Resume _BeginSc ____EndSc _____Exit _Elapsed
   user1  manager 10:45:33                    10:45:33 10:51:10 10:51:10  2:54.39
   user2   typist 10:45:48                    10:45:48 10:51:26 10:51:26  2:38.50
   user3  manager 10:46:03                    10:46:03 10:51:41 10:51:42  2:23.12
   user4  manager 10:46:18 10:53:38 10:53:46                                18.44
   user5   typist 10:46:33                    10:46:33 10:52:11 10:52:11  1:53.86
   user6  manager 10:46:48                    10:46:48 10:52:26 10:52:26  1:38.63
   user7  manager 10:47:03                    10:47:03                    7:02.11
   user8   typist 10:47:18                    10:47:18 10:52:56 10:52:56  1:08.41
```

Start	The time script execution began
Suspend	The time of the most recent Suspend() in the script
Resume	The time the script resumed execution after a Suspend()
BeginSc	The time of Beginscenario() execution, usually before any transactions are executed
EndSc	The time of Endscenario() execution
Exit	The time of script completion
Elapsed	The time elapsed since the most recent script activity (If a script is exited, it will indicate the time since exiting. If a script has begun a scenario, as shown above, it will indicate the time elapsed since the scenario was started.)

6.9.1.6 EMPOWER Monitor View 6

View 6 of `mon` displays function response times as functions complete. With this view, users often can determine how the SUT is handling the workload.

```
Mon Feb 11 10:51:11          EMPOWER MONITOR V3.1.8     (c) PERFORMIX, Inc. 1995
View 6 of 7  Script 1 of 8           Running      CurrFnRt Sorting v  Interval 1

ScriptId __Script ____NFn _AveFnRt _______LastFn LastFnRt _______CurrFn CurrFnRt
   user7  manager                                                  vi  2:43.59
   user2   typist       2  1:27.23           vi   1:26.45          vi  1:26.09
   user3  manager       2  1:27.08           vi   1:26.14          vi  1:11.20
   user5   typist       2  1:26.98           vi   1:25.94          vi    40.00
   user6  manager       2  1:27.06           vi   1:26.01          vi    25.22
   user8   typist       2  1:26.90           vi   1:26.00
   user4  manager       1  3:11.00           vi   3:11.00
   user1  manager       3  1:27.47           vi   1:28.19
```

`NFn`	The number of functions completed (Each function is a set of transactions.)
`AveFnRt`	Average response time of functions completed
`LastFn`	The last function completed (`LastFn` and `CurrFn` will be displayed only when you specify a `BeginFunction()` and `EndFunction()` pair.)
`LastFnRt`	The response time of the last function completed
`CurrFn`	The function currently executing
`CurrFnRt`	The response time of the function currently executing

6.9.1.7 EMPOWER Monitor View 7

View 7 of `mon` displays transaction response times as transactions complete.

```
Mon Feb 11 10:52:10          EMPOWER MONITOR V3.1.8     (c) PERFORMIX, Inc. 1995
View 7 of 7  Script 1 of 8           Running        CurrXr Sorting v  Interval 5

ScriptId __Script ____NXr _AveXrRt _______LastXr LastXrRt _______CurrXr CurrXrRt
   user7  manager       1      .13            vi      .13            who
   user8   typist      28      .46            vi      .13             vi
   user6  manager      30      .48            vi      .09             vi      .56
   user4  manager                                                   date
   user5   typist      33      .49            ls      .92
   user3  manager      33      .51            ls      .92
   user2   typist      33      .50            ls      .92
   user1  manager      33      .51            ls      .95
```

`NXr`	The number of transactions `Xmit()-Rcv()` pairs completed
`AveXrRt`	The average response time of transactions completed
`LastXr`	The last transaction completed (`LastXr` and `CurrXr` will be displayed only when you specify a `BeginTransaction()` and `EndTransaction()` pair.)
`LastXrRt`	The response time of the last transaction completed
`CurrXr`	The transaction currently executing
`CurrXrRt`	The response time of the transaction currently executing

6.9.2 Monitor Views for EMPOWER/X

The ten views available in the Monitor for EMPOWER/X (xmon) are described in this section.

6.9.2.1 EMPOWER/X Monitor View 1

View 1 of xmon displays the state of each script and the data transmitted and received by each script.

```
Fri Apr 16 16:41:51         EMPOWER/X MONITOR V4.1.4   (c) PERFORMIX, Inc. 1995
View  1 of 10  Script 2 of 4        Running       ScriptId Sorting ^  Interval 5

ScriptId __Script _____State Cl ______________LastXmit __________________LastRcv
      u1  script1  keystring  2 Mls^Mdate^Mwho^Mpwd^Mt /Xstuff^J[empower@nomad]
      u2  script1    textrcv  2                  ^Mls^M erycode.Jeff^Jquerycode.S
      u3  script1 disconnect  1             <LEFT_ALT>             PPPPPPPPPPPPP
      u4  script1    textrcv  2     ^Mls^Mdate^Mwho^M 6 10:18^J[empower@nomad]
```

ScriptId	A unique identifier specified in the Mix script table (If you run a script from the command line, the ScriptId will be the same as the script name.)
Script	The name of the compiled script (The source version of the script is typically found in script.c.)
State	The current state of the script's execution (keystring, textrcv, suspend, think, or exit)
Cl	The active client (application) on the SUT
LastXmit	The last characters transmitted to the SUT by the script (Characters scroll from right to left as they are transmitted so that the rightmost character is the last character transmitted.)
LastRcv	The last characters received from the SUT by the script (Characters scroll from right to left as they are received so that the rightmost character is the last character received.)

Monitor ensures that `ScriptIds` are unique by allowing a script to "steal" the Monitor slot of another script having the same `ScriptId`. If you run a load test again after the first run has completed, the second run will "re-use" the previous load test's Monitor slots. Users must be careful to designate different script IDs.

Sorting by `State` is useful when most scripts are in the same state or entering the same state, as when scripts are exiting or suspending.

When many control characters are displayed, entering '.' (dot) mode allows more characters to be displayed in the `LastXmit` and `LastRcv` fields .

09597494.102600

6.9.2.2 EMPOWER/X Monitor View 2

View 2 of xmon displays detailed information of script execution states, idle time, and the number of warning and timeout messages for each script. This view often is used for debugging scripts.

```
Fri Apr 16 16:41:23          EMPOWER/X MONITOR V4.1.4    (c) PERFORMIX, Inc. 1995
View  2 of 10  Script 1 of 4         Running       ScriptId Sorting ^  Interval 5

ScriptId __Script _____State Cl _____StatePattern __Idle NTo Limit __ToCondition
      u1  script1   internal  2       QueryColors   1.67        10   10 continue
      u2  script1   internal  1       GrabPointer    .53        10   10 continue
      u3  script1    textrcv  2   ConfigureWindow    .87        10   10 continue
      u4  script1   internal  2       QueryColors   1.67        10   10 continue
```

StatePattern	Additional information relative to the script execution state, such as the string being transmitted (keystring), the string being looked for (rcv), the current think value (think), or the exit status (exit)
Idle	The time, in seconds, that the script has been idle, or the time since the script last updated the shared memory segment
NTo	The number of EMPOWER/X timeouts issued to the script (i.e. when an expected rcv string never arrives)
Limit	The limit of the number of EMPOWER/X warnings and timeouts issued before script termination
ToCondition	The number of seconds before a timeout will occur and the action to be taken upon timeout

6.9.2.3 EMPOWER/X Monitor View 3

View 3 of `xmon` displays client information for each script, including the current client and the last messages received from the client and the X server.

```
Fri Apr 16 16:41:29          EMPOWER/X MONITOR V4.1.4   (c) PERFORMIX, Inc. 1995
View  3 of 10  Script 2 of 4        Running       ScriptId Sorting ^  Interval 5

ScriptId __Script __Client Cl NCl __Seq ________LastSMsg _______________LastCMsg
      u1  script1           2   2    45  ConfigureNotify         ConfigureWindow
      u2  script1           2   2    73          FocusIn   TranslateCoordinates
      u3  script1           2   2   165         NoExpose              ImageText8
      u4  script1           2   2    57            Reply             GetGeometry
```

`NCl`	The number of clients in the script
`Seq`	The script location (line number) of the current message
`LastSMsg`	The last message (event) sent from the X server to the client
`LastCMsg`	The last message (request) received from the X client on the SUT

6.9.2.4 EMPOWER/X Monitor View 4

View 4 of `xmon` displays the script execution completion percentage as well as mouse and display information.

```
Fri Apr 16 16:41:33          EMPOWER/X MONITOR V4.1.4    (c) PERFORMIX, Inc. 1995
View  4 of 10  Script 2 of 4        Running       ScriptId Sorting ^  Interval 5

ScriptId __Script _Line NLine _Pct _______Segment ____X ____Y ___________Display
      u1  script1   101   223 45.3                  325   164
      u2  script1   135   223 60.5                  325   164
      u3  script1   167   223 74.9                  325   164
      u4  script1   101   223 45.3                  325   164
```

`Line`	The line in the script source (.c) file currently executing
`NLine`	The number of lines in the source (.c) file
`Pct`	The percentage of script completion
`Segment`	The segment currently executing
`X`	The x-coordinate of the mouse pointer
`Y`	The y-coordinate of the mouse pointer
`Display`	The name of the X-window where the executing script is displayed

6.9.2.5 EMPOWER/X Monitor View 5

View 5 of `xmon` displays general script information such as the script's process ID, the owner of the script, think time distribution, and user-defined notes.

```
Fri Apr 16 16:41:35          EMPOWER/X MONITOR V4.1.4   (c) PERFORMIX, Inc. 1995
View  5 of 10  Script 2 of 4         Running       ScriptId Sorting ^  Interval 5

ScriptId __Script __Pid ___Owner Conn _________Log Type _______ThinkTime _____Note
      u1  script1  1070  empower    1          log1  5.0    constant 2.0
      u2  script1  1071  empower    2          log2  5.0    constant 2.0
      u3  script1  1076  empower    3          log3  5.0    constant 2.0
      u4  script1  1079  empower    4          log4  5.0    constant 2.0
```

`Pid`	The process ID of the script
`Owner`	The log in ID of the owner of the script (i.e. the person who started the script)
`Conn`	The connection number used by the executing script
`Log`	The log file to which the script is writing
`Type`	The typing speed of the script
`ThinkTime`	The think time distribution and parameters
`Note`	A user-defined note placed in the script by using the `Note()` function (A note can be placed at any location in the script and changed as appropriate.)

6.9.2.6 EMPOWER/X Monitor View 6

View 6 of `xmon` displays times of various events as well as elapsed time. Users often can estimate load test completion time by looking at this view.

```
Fri Apr 16 16:41:37        EMPOWER/X MONITOR V4.1.4    (c) PERFORMIX, Inc. 1995
View  6 of 10  Script 2 of 4       Running       ScriptId Sorting ^  Interval 5

ScriptId __Script ___Start _Suspend __Resume _BeginSc ___EndSc ____Exit _Elapsed
      u1  script1 16:40:54                        .90                      41.90
      u2  script1 16:40:54                        .33                      41.94
      u3  script1 16:40:56                        .51                      39.64
      u4  script1 16:40:58                        .30                      38.19
```

`Start`	The time script execution began
`Suspend`	The time of the most recent `Suspend()` in the script
`Resume`	The time the script resumed execution after a `Suspend()`
`BeginSc`	The time of `Beginscenario()` execution, usually before any transactions are executed
`EndSc`	The time of `Endscenario()` execution
`Exit`	The time of script completion
`Elapsed`	The time elapsed since the most recent script activity (If a script exited, it will indicate the time since exiting. If a script has begun a scenario, it will indicate the time elapsed since the scenario was started.)

6.9.2.7 EMPOWER/X Monitor View 7

View 7 of `xmon` displays function response times as functions complete. With this view, users often can determine how the SUT is handling the workload.

```
Fri Apr 16 16:41:39          EMPOWER/X MONITOR V4.1.4    (c) PERFORMIX, Inc. 1995
View  7 of 10  Script 2 of 4        Running        ScriptId Sorting ^  Interval 5

ScriptId __Script ____NFn _AveFnRt _______LastFn LastFnRt _______CurrFn CurrFnRt
      u1  script1                                          shell_cmds      2.63
      u2  script1                                          shell_cmds      9.91
      u3  script1                                          shell_cmds     13.51
      u4  script1                                          shell_cmds      4.30
```

`NFn`	The number of functions completed (Each function is a set of transactions.)
`AveFnRt`	Average response time of functions completed
`LastFn`	The last function completed (`LastFn` and `CurrFn` will be displayed only when you specify a `BeginFunction()` and `EndFunction()` pair.)
`LastFnRt`	The response time of the last function completed
`CurrFn`	The function currently executing
`CurrFnRt`	The response time of the function currently executing

6.9.2.8 EMPOWER/X Monitor View 8

View 8 of xmon displays transaction response times as transactions complete.

```
Fri Apr 16 16:41:41         EMPOWER/X MONITOR V4.1.4   (c) PERFORMIX, Inc. 1995
View  8 of 10  Script 2 of 4        Running        ScriptId Sorting ^  Interval 5

ScriptId __Script ____NXr _AveXrRt _______LastXr LastXrRt _______CurrXr CurrXrRt
      u1  script1       1      .32        return      .32          date     2.44
      u2  script1       1      .41        return      .41
      u3  script1       4     1.75           pwd      .88           tty     7.52
      u4  script1       2     1.55          date     2.69           who     2.09
```

`NXr`	The number of transactions `KeyString()-TextRcv()` pairs completed (e.g. data base queries, UNIX shell commands)
`AveXrRt`	The average response time of transactions completed
`LastXr`	The last transaction completed (`LastXr` and `CurrXr` will be displayed only when you specify a `BeginTransaction()` and `EndTransaction()` pair.)
`LastXrRt`	The response time of the last transaction completed
`CurrXr`	The transaction currently executing
`CurrXrRt`	The response time of the transaction currently executing

Note: `NXr`, `AveXrRt`, `LastXrRt`, and `CurrXrRt` will have values only when a `BeginTransaction()-EndTransaction()` pair has been specified.

6.9.2.9 EMPOWER/X Monitor View 9

View 9 of `xmon` displays statistics that indicate the number of events and other messages sent by the X clients and X servers.

```
Fri Apr 16 16:41:45          EMPOWER/X MONITOR V4.1.4   (c) PERFORMIX, Inc. 1995
View  9 of 10  Script 2 of 4        Running       ScriptId Sorting ^  Interval 5

ScriptId __Script ________NR ________Nr ________Ne ________No ______NCMsgs _____NSMsgs
      u1  script1      891        96       130         0        891        226
      u2  script1      868        96       125         0        868        221
      u3  script1      941        96       145         0        941        241
      u4  script1      929        96       138         0        929        234
```

NR	The number of Request (R) messages sent by the X client(s)
Nr	The number of reply (r) messages sent by the X server
Ne	The number of event (e) messages sent by the X server
No	The number of error (o) messages sent by the X server
NCMsgs	The number of messages sent by the X client(s)
NSMsgs	The number of messages sent by the X server(s)

6.9.2.10 EMPOWER/X Monitor View 10

View 10 of xmon displays the number of bytes of messages sent by the X clients and the X servers.

```
Fri Apr 16 16:41:48          EMPOWER/X MONITOR V4.1.4    (c) PERFORMIX, Inc.
1995
View 10 of 10  Script 2 of 4        Running       ScriptId Sorting ^  Interval 5

ScriptId __Script __NRBytes __NrBytes __NeBytes __NoBytes ____NCBytes ___NSBytes
      u1  script1     23928     11148      4224         0       23928      15372
      u2  script1     23232     11148      4000         0       23232      15148
      u3  script1     25476     11148      4864         0       25476      16012
      u4  script1     24608     11148      4416         0       24608      15564
```

NRBytes	The number of bytes of the Request (R) messages sent by the X client(s)
NrBytes	The number of bytes of the reply (r) messages sent by the X server
NeBytes	The number of bytes of the event (e) messages sent by the X server
NoBytes	The number of bytes of the error (o) messages sent by the X server
NCBytes	The number of bytes of the messages sent by the X client(s)
NSBytes	The number of bytes of the messages sent by the X server(s)

6.9.3 EMPOWER/CS Monitor Views

The seven views available for the Monitor for EMPOWER/CS (`csmon`) are described in this section.

6.9.3.1 EMPOWER/CS Monitor View 1

View 1 of `csmon` displays the state of each script and the data transmitted and received by each script.

```
Wed Jan  4 15:33:43       EMPOWER/CS MONITOR V1.0.1     (c) PERFORMIX, Inc. 1995
View 1 of 7  Script 1 of 4          Running       ScriptId Sorting ^  Interval 5

ScriptId __Script State ___________________________LastEvent _____CurrentWindow
   user1      foo  type              <SysKeyPress>James Smith          Employee
   user2      foo event         <SysKeyPress><LeftButtonPress>              Run
   user3      foo event                            <SysKeyPress>            Help
   user4      foo  type               <SysKeyPress>test.dat^M           Save As
```

`ScriptId`	A unique identifier specified in the script table and used by the Mix tool (If you run a script from the command line, the `ScriptId` will be the same as the script name.)
`Script`	The name of the compiled script (The source version of the script typically is found in a `.c` file)
`State`	The current state of the script's execution
`LastEvent`	The last user interactions transmitted to the database server by the script (Characters scroll from right to left as they are transmitted so that the rightmost character is the last character transmitted.)
`CurrentWindow`	The last `CurrentWindow()` function.

Monitor enforces that `ScriptIds` are unique by allowing a script to "steal" the Monitor slot of another script having the same `ScriptId`. If you run one load test after the first is completed, the second run will "re-use" the first Monitor slots. Users must be careful to designate different script IDs.

Sorting by State is useful when most scripts are in the same state or entering the same state, as when scripts are exiting or suspending.

6.9.3.2 EMPOWER/CS Monitor View 2

View 2 of csmon displays detailed information of script execution states, idle time, and the number of database errors and timeout messages for each script. This view often is used for debugging scripts.

```
Wed Sep 28 14:09:40     EMPOWER/CS MONITOR V1.0.1   (c) PERFORMIX, Inc. 1995
View 2 of 7  Script 1 of 8         Running         Idle Sorting v  Interval 1

ScriptId__Script State _______StatePattern ___Idle NDBerror NTo NExec NRFetch
     foo      foo  wrcv      SfCoCwAcSfDwPt    1.08
```

`StatePattern`	Additional information relative to the script execution state
`Idle`	The time, in seconds, that the script has been idle or the time since the shared memory segment was last updated by the script
`NDBerror`	The number of database errors issued to the script
`NTo`	The number of EMPOWER timeouts issued to the script (i.e., when an expected `WindowRcv()` never arrives)
`NExec`	The number of `Exec()` statements in the script that were executed,
`NRFetch`	The number of fetches from the database after the `Exec()`

6.9.3.3 EMPOWER/CS Monitor View 3

View 3 of `csmon` displays script execution completion percentage, the script source file name, the log file name, the data type used by the script, and amount of think time.

```
Wed Sep 28 14:09:40     EMPOWER/CS MONITOR V1.0.1   (c) PERFORMIX, Inc. 1995
View 3 of 7  Script 1 of 4       Running      ScriptId Sorting ^  Interval 5

ScriptId _Script Line NLine _Pct _____Source ________Log Type_____Thinktime
     foo     foo   33   63 52.4         foo         foo 5.0 uniform 1.0 2.5
```

`Line`	The line in the script source (.c) file currently executing
`NLine`	The number of lines in the source (.c) file
`Pct`	The percentage of script completion
`Source`	The name of the script source file (may be a function source file)
`Log`	The log file to which the script is writing
`Type`	The typing speed of the script
`Thinktime`	The think time distribution and parameters

6.9.3.4 EMPOWER/CS Monitor View 4

View 4 of `csmon` displays general information about the scripts.

```
Wed Sep 28 14:09:40         EMPOWER/CS MONITOR V1.0.1   (c) PERFORMIX, Inc. 1995
View 4 of 7  Script 1 of 4          Running       ScriptId Sorting ^  Interval 5

ScriptId __Script __Pid ___Display____________Arguments __ToCondition _____Note
     foo       foo   704       ergy                        300 continue
```

`Pid`	The process ID of the script
`Display`	The display terminal, if applicable
`Arguments`	The number of arguments to the scripts
`ToCondition`	The number of seconds before a timeout will occur and the action to be taken upon timeout
`Note`	A user-defined note placed in the script by using the `Note()` function (A note can be placed at any location in the script and changed as appropriate.)

6.9.3.5 EMPOWER/CS Monitor View 5

View 5 of `csmon` displays times of various events as well as elapsed time information. Users often can estimate load test completion time by looking at this view.

```
Wed Sep 28 14:09:40        EMPOWER/CS MONITOR V1.0.1   (c) PERFORMIX, Inc. 1995
View 5 of 7  Script 4 of 8          Running      ScriptId Sorting ^  Interval 1

ScriptId __Script ____Start _Suspend __Resume _BeginSc ____EndSc _____Exit _Elapsed
   user1  manager 10:45:33                    10:45:33 10:51:10 10:51:10  2:54.39
   user2   typist 10:45:48                    10:45:48 10:51:26 10:51:26  2:38.50
   user3  manager 10:46:03                    10:46:03 10:51:41 10:51:42  2:23.12
   user4  manager 10:46:18 10:53:38 10:53:46                                18.44
   user5   typist 10:46:33                    10:46:33 10:52:11 10:52:11  1:53.86
   user6  manager 10:46:48                    10:46:48 10:52:26 10:52:26  1:38.63
   user7  manager 10:47:03                    10:47:03 -                  7:02.11
   user8   typist 10:47:18                    10:47:18 10:52:56 10:52:56  1:08.41
```

`Start`	The time script execution began
`Suspend`	The time of the most recent `Suspend()` in the script
`Resume`	The time the script resumed execution after a `Suspend()`
`BeginSc`	The time of `Beginscenario()` execution, usually before any transactions are executed
`EndSc`	The time of `Endscenario()` execution
`Exit`	The time of script completion
`Elapsed`	The time elapsed since the most recent script activity (If a script has exited, the time since exiting will be indicated. If a script has begun a scenario, as shown above, the time elapsed since the scenario was started will be indicated.)

6.9.3.6 EMPOWER/CS Monitor View 6

View 6 of `csmon` displays function response times as functions complete. With this view, users often can determine how the SUT is handling the workload.

```
Wed Sep 28 14:09:40        EMPOWER/CS MONITOR V1.0.1   (c) PERFORMIX, Inc. 1995
View 6 of 7  Script 1 of 8           Running      CurrFnRt Sorting v  Interval 1

ScriptId __Script ____NFn _AveFnRt _______LastFn LastFnRt _______CurrFn CurrFnRt
   user7  manager                                                   vi  2:43.59
   user2   typist       2  1:27.23            vi  1:26.45           vi  1:26.09
   user3  manager       2  1:27.08            vi  1:26.14           vi  1:11.20
   user5   typist       2  1:26.98            vi  1:25.94           vi    40.00
   user6  manager       2  1:27.06            vi  1:26.01           vi    25.22
   user8   typist       2  1:26.90            vi  1:26.00
   user4  manager       1  3:11.00            vi  3:11.00
   user1  manager       3  1:27.47            vi  1:28.19
```

`NFn`	The number of functions completed (Each function is a set of transactions.)
`AveFnRt`	Average response time of functions completed
`LastFn`	The last function completed (`LastFn` and `CurrFn` will be displayed only when you specify a `BeginFunction()` and `EndFunction()` pair.)
`LastFnRt`	The response time of the last function completed
`CurrFn`	The function currently executing
`CurrFnRt`	The response time of the function currently executing

6.9.3.7 EMPOWER/CS Monitor View 7

View 7 of `csmon` displays transaction response times as transactions complete.

```
Wed Sep 28 14:09:40     EMPOWER/CS MONITOR V1.0.1    (c) PERFORMIX, Inc. 1995
View 7 of 7  Script 1 of 8         Running        CurrXr Sorting v  Interval 5

ScriptId __Script ____NXr _AveXrRt _______LastXr LastXrRt _____CurrXr CurrXrRt
   user7  manager       1      .13            vi      .13        who
   user8   typist      28      .46            vi      .13         vi
   user6  manager      30      .48            vi      .09         vi      .56
   user4  manager                                               date
   user5   typist      33      .49            ls      .92
   user3  manager      33      .51            ls      .92
   user2   typist      33      .50            ls      .92
   user1  manager      33      .51            ls      .95
```

`NXr`	The number of transactions completed
`AveXrRt`	The average response time of transactions completed
`LastXr`	The last transaction completed
`LastXrRt`	The response time of the last transaction completed
`CurrXr`	The transaction currently executing
`CurrXrRt`	The response time of the transaction currently executing

[This page intentionally left blank]

7.0 EMPOWER/GV

As an advanced feature of the EMPOWER products, the EMPOWER/GV tool provides additional control over multi-user emulations by creating and controlling global variables that are shared among executing scripts. Variables may be defined to direct scripts to terminate gracefully when they run an indefinite process, to act as a counter for scripts that must perform a fixed amount of work before terminating, or to synchronize the execution of multiple scripts.

Controlling global variables is useful during a multi-user emulation when one script depends on another script's actions to execute properly. For example, several scripts may delete records from a single database. A global variable can be assigned to tell each script which record to delete. Because global variables retain their value when script execution has completed, these same scripts can be executed several times without resetting the database.

Global variables also may be used to control access to a commonly used database or document. For example, several scripts may have to edit a common word processing document. If multiple scripts attempt to edit the same document, problems may occur. You can define a global variable to protect a document while one script edits it. Other scripts attempting to edit the document must wait until the first script has finished and the global variable "unprotects" the document.

The Global Variable library permits variables to be shared among scripts. Access to and manipulation of global variables is provided at the UNIX driver's shell prompt by Global Variable commands and from within scripts by Global Variable functions. Commands can create and remove variables. Both commands and functions are used to assign values to variables, read the value of a variable, test the value of a variable, and protect a variable.

Note: A knowledge of C language is useful when using Global Variables because script functions that access global variables are actually C language statements.

7.1 Additional Installation and Environment Setup

The EMPOWER/GV software is installed automatically when the EMPOWER, EMPOWER/X, or EMPOWER/CS software is installed. After an EMPOWER product has been installed successfully, the following steps will complete EMPOWER/GV installation (*Note:* These steps assume you install the software in `/usr/empower`):

Log in to your system as the user who owns the EMPOWER software. This user typically is `empower` or `root`.

Set your shell environment variable equal to the directory in which the EMPOWER software was installed.

If you are a Bourne-shell user, execute:

```
$ EMPOWER=/usr/empower
$ export  EMPOWER
```

If you are a C-shell user, execute:

```
$ setenv  EMPOWER  /usr/empower
```

Then, execute the following command:

```
$ EMPOWER/bin/gvinstall
```

If the `ln` command on your system allows symbolic linking, you can save some disk space by executing:

```
$ EMPOWER/bin/gvinstall  -s
```

7.2 Architecture

EMPOWER/GV consists of three parts:

- *Global Variables* which are created and stored in the shared memory of the UNIX script driver
- *Global Variable Commands* which are executed from the shell prompt on the UNIX script driver to access variables
- *Global Variable Functions* which are placed in the script .c file to allow a script to access a variable during script execution

Global variables are created by a Global Variable command (`gv_init`) and stored in the shared memory of the UNIX script driver. This shared memory is accessible by both Global Variable commands and functions. A variable remains in shared memory until it is removed and it will retain its value after script execution. If the UNIX script driver is rebooted, all global variables are removed. Since global variables are stored in shared memory, your system must have the System V Shared Memory Facility installed to use the Global Variables tool.

You can enter Global Variable commands at the shell prompt of the UNIX script driver terminal. These commands also may be used during script execution with the Mix tool. Global Variable functions are placed in a script .c file during editing and can be used only when the script is compiled with Scc, Xscc, or Cscc. (*Note:* For EMPOWER or EMPOWER/X users, scripts executed with Preview or Xpreview will not recognize the Global Variable functions.)

Global Variable commands and functions are used to create, allocate, read, update, test, and protect global variables. Changes to the value of a variable are made immediately. You can update a variable by assigning a new value or by applying a mathematical operation to the current value of the variable. You may test the value of a variable and use it in conjunction with the C language `while` statement so that an operation continues while the variable has a certain value. You also may protect a variable with a Global Variable command or function so that it can be accessed only by the script that protected it.

7.3 Global Variable Commands

Global Variable commands are UNIX shell commands used from the shell prompt on the UNIX script driver and from shell scripts to control global variables. They control access to a variable, read the current value of a variable, specify a new value for a variable, test the value of a variable, and control the shared memory segment used by global variables. All Global Variable commands begin with "gv_."

When a Global Variable command accesses a variable, the operation completes before any other command or function may access the variable. (*Note:* This uninterrupted operation is referred to as "atomic" referencing.)

When a Global Variable command (except for `gv_test`) executes successfully, the return code is set to zero. If a command results in an error, an error message is printed to the standard error output destination, or `stdout`, which is usually the UNIX script driver. With the exception of the `gv_test` command, an error will set the return code to one.

If you enter a command with an improper number of arguments, a usage message displays the correct syntax of the command.

Syntax help for each command can be obtained by entering the command name followed by a hyphen:

```
$ gv_add  -
Usage:
        gv_add name value
$
```

7.3.1 Valid Test Relations

Valid test relations used for relational comparisons in Global Variable commands are listed below:

Equality Relations

Relation String	Relation Test
"=="	variable equal to value
"!="	variable NOT equal to value
"<"	variable less than value
"<="	variable less than or equal to value
">"	variable greater than value
">="	variable greater than or equal to value

Bit-Wise Relations

Relation String	Relation Test
"&"	variable AND value
"!&"	variable NOT AND value
"|"	variable OR value
"!|"	variable NOT OR value
"^"	variable EXCLUSIVE OR value
"!^"	variable NOT EXCLUSIVE OR value

Logical Relations

Relation String	Relation Test
"?"	variable is non-zero (non-NULL for str)
"!"	variable is zero (NULL for str)

7.3.2 Valid Variable Types

Many variable types are supported by the Global Variable Library. The name of the type is used as a parameter in the `gv_init` command. EMPOWER/GV includes support for type name aliases which allows a shorter type name to be used.

The valid types, aliases, and descriptions of variable types are listed in the following table:

Type Name	Alias	Description
`char`	`[none]`	character
`unsigned char`	`uchar`	unsigned character
`int`	`[none]`	integer
`unsigned int`	`uint, unsigned`	unsigned integer
`short int`	`short`	short integer
`unsigned short`	`intushort, ushort int, unsigned short`	unsigned short integer
`long int`	`long`	long integer
`unsigned long int`	`ulong, ulong int, unsigned long`	unsigned long integer
`float`	`[none]`	single precision floating point
`double`	`long float`	double precision floating point
`string`	`str`	character string
`parallel`	`par`	parallel global variable

The maximum length of a string is 32 characters.

Variable type "`parallel`" refers to parallel Global Variables. Parallel global variables allow a specified number of scripts in a multi-user emulation to execute a series of transactions at the same time. All other scripts in the emulation will block while the first scripts are working in parallel. The blocked scripts wait until the parallel scripts have completed execution.

Note that only the following commands may access Global Variables of type "parallel": `gv_init`, `gv_setparallel`, `gv_getparallel`, `gv_parallel`, `gv_unparallel`, `gv_stat`, and `gv_rm`.

You must initialize the global variable as `parallel` type and then set it to a certain number of users (scripts) that will execute a series of transactions in parallel. You are allowed to set up to five parallel global variables and they must be initialized before you initialize any other global variables.

7.3.3 Access Control

The Global Variable commands for access control are `gv_init`, `gv_rm`, `gv_protect`, and `gv_unprotect`. These commands are described below:

`gv_init`	Creates the variable with the specified name, type, and initial value (All parallel variables should be initialized before any other variables.)
`gv_rm`	Removes a variable from shared memory (Variables are automatically removed if the system is rebooted.)
`gv_protect`	Prevents scripts from accessing a variable until the `gv_unprotect` command is entered
`gv_unprotect`	Removes protection from a variable, allowing scripts to access the variable

Errors will occur if you specify the variable type incorrectly, if a variable cannot be created, or if you try to remove a variable that does not exist.

7.3.4 Reading a Variable

The Global Variable commands for reading a variable are `gv_read`, `gv_getparallel`, and `gv_stat`.

The gv_read command returns the current value of the specified variable to the standard output destination (which is usually the terminal). In the following example, 3 is the current value of users.

```
$ gv_read  users
3
```

The gv_getparallel command returns the current value of the specified parallel variable.

Example:

```
$ gv_getparallel  workers
5
```

The gv_stat command sends to the standard output destination a table showing the name, type, and value of the specified variable. The table also shows the number of scripts that have allocated the variable and identifies the variable's protector. For example:

```
$ gv_stat  users
EMPOWER/GV V3.1.8, Serial#R00000-000, Copyright PERFORMIX, Inc. 1988-95
 Name              Type                   Value       Allocated Protector
 ----------------- ------------------- ----------- --------- ---------
 users             int                          3          0      NONE
```

If gv_stat is executed without an argument, information will be listed for all variables:

```
$ gv_stat
EMPOWER/GV V3.1.8, Serial#R00000-000, Copyright PERFORMIX, Inc. 1988-95
 Name               Type                Value       Allocated Protector
 ------------------ ------------------ ----------- --------- ----------
 cust               int                        100          0 CONSOLE
 para               parallel                     6          0
 foo                int                          1          1
```

7.3.5 Updating a Variable

The Global Variable commands for updating a variable are listed below:

`gv_write`	Assigns a new value to the variable
`gv_setparallel`	Assigns a new value to the parallel variable
`gv_inc`	Increases the value of the variable by one
`gv_dec`	Decreases the value of the variable by one
`gv_unparallel`	Increases the value of the parallel variable by one
`gv_add`	Adds the specified amount to the current value of the variable
`gv_sub`	Subtracts the specified amount from the current value of the variable
`gv_mul`	Multiplies the current value of the variable by the specified amount
`gv_div`	Divides the current value of the variable by the specified amount
`gv_mod`	Performs a modulo operation on the variable
`gv_lshift`	Performs a bit-wise shift to the left on the variable
`gv_rshift`	Performs a bit-wise shift to the right on the variable
`gv_and`	Applies bit-wise AND masking to the variable
`gv_or`	Applies bit-wise OR masking to the variable
`gv_xor`	Applies bit-wise EXCLUSIVE-OR masking to the variable

Each of these commands returns the current value of the variable to the standard output destination before performing the specified operation. For example, suppose the variable `users` has a current value of 5 and you want to make the value 6, then change the value by subtracting 4. This interaction would occur as demonstrated below:

```
$ gv_write  users  6
5
$ gv_sub  users  4
6
$ gv_read  users
2
$
```

7.3.6 Testing a Variable

The Global Variable commands for testing a variable are `gv_test`, `gv_waitwhile`, `gv_waituntil`, and `gv_parallel`. These commands compare the current value of a variable to a specified value or condition.

The valid test criteria, specified in Section 7.3.1, must be either equality, bit-wise, or logical relations. For example, the equality relation ">=" tests if the variable value is greater than or equal to the specified comparison value. The equality and bit-wise relations require that the command include three parameters: the variable name, the relation, and a comparison value. Logical relations only require the variable name and the relation.

The `gv_test` command simply tests the variable as specified. Results of the `gv_test` command are listed below:

If Comparison	Print to Standard Output	Set the Return Code to
is true	1	0
is false	0	1
fails due to error	error message	2

Example:

```
$ gv_read  users
2
$ gv_test  users  "=="  2
1
$ echo  $status
0
$ gv_inc  users
2
$ gv_read  users
3
$ gv_test  users  "=="  2
0
$ echo  $status
1
$
```

Note: `$?` replaces `$status` when displaying the return value for the Bourne shell and the Korn shell.

The `gv_waitwhile` command waits while the test condition is true. The `gv_waituntil` command waits until the test condition is true. These commands do not send a message to the standard output destination. If the command is successful, the return code is set to zero. If the variable cannot be referenced, it is set to one. The difference between `gv_waitwhile` and `gv_waituntil` is illustrated below.

`gv_waitwhile("a", "<", 3)`	`gv_waituntil("a", "<", 3)`
results in:	*results in:*
a = 0 -> wait	a = 0 -> do not wait
a = 1 -> wait	a = 1 -> do not wait
a = 2 -> wait	a = 2 -> do not wait
a = 3 -> do not wait	a = 3 -> wait
a = 4 -> do not wait	a = 4 -> wait
a = 5 -> do not wait	a = 5 -> wait

The gv_parallel command waits for a parallel variable to become greater than zero then decrements the value of the variable by one.

7.3.7 Shared Memory Segment Control

Whenever a Global Variable Command is executed, the necessary shared memory segment is created if it does not already exist. The created shared memory segment is large enough to support 128 global variables by default. The gv_seg command can create a shared memory segment to support fewer or more global variables. This command also can remove an existing shared memory segment which removes all existing global variables.

For example, to create a shared memory segment that supports 150 global variables, use the following command (assuming a shared memory segment does not already exist):

```
$ gv_seg  150
```

If a shared memory segment already exists, which would be the case if you already had executed a Global Variables Command, it must be removed with the -r option of gv_seg before a new segment is created:

```
$ gv_seg  -r
$ gv_seg  150
```

Use caution when removing an existing shared memory segment with gv_seg -r. Since removing the shared memory segment removes all global variables, this command should not be executed while global variable scripts are executing .

7.4 Global Variable Functions

Global Variable functions act as C language statements to access and control variables used by multiple scripts. Scripts with Global Variable functions must be compiled with Scc, Xscc, or Cscc before they can be executed. (*Note:* For EMPOWER or EMPOWER/X users, scripts executed with Preview or XPreview will not recognize Global Variable functions.)

Global Variable functions provide features similar to Global Variable commands. They control access to a variable, read the current value of a variable, specify a new value for a variable, and test the value of a variable. All Global Variable functions begin with "Gv_."

When a Global Variable function accesses a variable, the operation completes before any other command or function can access the variable. (*Note:* This uninterrupted operation is referred to as "atomic" referencing.)

If a Global Variable function can not execute because an error occurred, an error message is sent to the standard error destination and the script will exit.

The following error messages occur for any function when the specified variable has not been allocated to the script or when the variable does not exist:

```
Attempt to access a global variable that is not allocated
Global variable not allocated to this script
Global variable does not exist
```

7.4.1 Access Control

The Global Variable functions for access control are `Gv_alloc()`, `Gv_protect()`, and `Gv_unprotect()`.

These commands are described below:

`Gv_alloc()`	Allocates access to a variable
`Gv_free()`	De-allocates access to a variable
`Gv_protect()`	Prevents other scripts from accessing a variable until the `Gv_unprotect()` function is executed
`Gv_unprotect()`	Removes protection from a variable, allowing other scripts to access the variable

If a script references a global variable, the `Gv_alloc()` function should be the first function in the script. The `Gv_alloc()` function has two parameters: variable name and variable type. The variable type must match the type specified when the variable was created with the `gv_init` command. For example:

```
Gv_alloc("users", "int");
```

You can de-allocate a global variable with `Gv_free()`. If a global variable is not de-allocated with `Gv_free()`, it automatically will de-allocate when the script exits.

If a script tries to de-allocate a variable that has not been allocated to the script, an error will occur. The following error messages occur for any function when the specified variable has not been allocated to the script or when the variable does not exist:

```
Global variable not currently allocated
Global variable does not exist
```

The parameter of the `Gv_protect()` and `Gv_unprotect()` functions is the name of the variable (e.g., `users` from the example above). Only the script that protects a variable may unprotect it.

7.4.2 Reading a Variable

The Global Variable functions for reading a variable are `Gv_read()`, `Gv_readv()`, `Gv_getparallel()`, and `Gv_stat()`. The `Gv_read()` and `Gv_readv()` functions return the current value of a variable. The `Gv_getparallel()` function returns the current value of the specified parallel variable. The `Gv_stat()` function supplies status information about variables.

The `Gv_read()` function returns the current value of the specified global variable. Generally, this value is stored in a new variable. For example:

```
int curcount;
Gv_alloc("count", "int");
curcount = Gv_read("count");
```

The value of `count` is read and stored in an integer variable called `curcount`.

The value returned by the `Gv_read()` function is always an integer. If you need to use the return value and the actual value of a global variable is not an integer, you must use the function name ending with a "v" to obtain the correct value. In this case, an additional parameter is used to indicate where to store the value.

For example, if you want a script to read the current value of a parameter called `balance` but the value is not an integer, use the `Gv_readv()` function as in the following script file excerpt:

```
float curbalance;
Gv_alloc("balance", "float");
Gv_readv("balance", &curbalance);
```

In this example, the `Gv_readv()` function retrieves the current value of the variable `balance` and stores it in the floating point variable `curbalance`.

7.4.3 Updating a Variable

The Global Variable functions for updating a variable are listed below:

`Gv_write()`, `Gv_writev()`	Assigns a new value to the variable
`Gv_setparallel()`	Assigns a new value to the parallel variable
`Gv_inc()`, `Gv_incv()`	Increases the value of the variable by one
`Gv_dec()`, `Gv_decv()`	Decreases the value of the variable by one
`Gv_unparallel()`	Increases the value of the parallel variable by one
`Gv_add()`, `Gv_addv()`	Adds the specified amount to the current value of the variable
`Gv_sub()`, `Gv_subv()`	Subtracts the specified amount from the current value of the variable
`Gv_mul()`, `Gv_mulv()`	Multiplies the current value of the variable by the specified amount
`Gv_div()`, `Gv_divv()`	Divides the current value of the variable by the specified amount
`Gv_mod()`, `Gv_modv()`	Performs a modulo operation on the variable
`Gv_lshift()`, `Gv_lshiftv()`	Performs a bit-wise shift to the left on the variable
`Gv_rshift()`, `Gv_rshiftv()`	Performs a bit-wise shift to the right on the variable
`Gv_and()`, `Gv_andv()`	Applies bit-wise AND masking to the variable
`Gv_or()`, `Gv_orv()`	Applies bit-wise OR masking to the variable
`Gv_xor()`, `Gv_xorv()`	Applies bit-wise EXCLUSIVE-OR masking to the variable

Functions without a trailing "v" are used for integer variables. Functions with a trailing "v" are used for non-integer variables when the current value of the variable is needed. The first parameter is the name of the variable for all update functions.

When an update function is executed, the original value of the variable is saved if the variable is an integer, or copied to a pointer location if the variable is not an integer. After the operation is performed and the resulting value is assigned, the original value of the variable is returned as an integer.

For example, to assign a new value to the variable `count`, use the following function if `count` is an integer:

```
Gv_write("count", 12);
```

To assign a new value to the non-integer variable `balance`, use:

```
float curbalance;
Gv_alloc("balance", "float");
Gv_writev("balance", 120.75, &curbalance);
```

Similarly, the following function can be used to increment the integer variable `count`:

```
Gv_inc("count");
```

The following statements could be used to increment the non-integer variable `balance`:

```
float curbalance;
Gv_alloc("balance", "float");
Gv_incv("balance", &curbalance);
```

The remaining update functions provide the same capabilities as the corresponding Global Variable commands. These functions require an additional parameter to specify the operand value.

In the following example, the value of the variable `count` is multiplied by four. The value of `count` prior to multiplication is stored in `curcount`:

```
int curcount;
Gv_alloc("count", "int");
curcount = Gv_mul("count", 4);
```

7.4.4 Testing a Variable

The Global Variable functions for testing a variable are `Gv_test()`, `Gv_waitwhile()`, `Gv_waituntil()`, and `Gv_parallel()`. These functions test the current value of a variable against a specified value or condition.

The valid test criteria (See Section 7.3.1) must be either equality, bit-wise, or logical relations. For example, the equality relation `">="` tests if the variable value is greater than or equal to the specified comparison value. The equality and bit-wise relations require that the function include three parameters: the variable name, the relation, and a comparison value. Logical relations only require the variable name and the relation.

The `Gv_test()` function simply tests the variable as specified. `Gv_test()` returns 1 if the test is true and 0 if the test is false. For example:

```
Gv_alloc("quitflag", "int");
if (Gv_test("quitflag", "==", 1))
     Exit(1);
```

The `Gv_waitwhile()` function waits while the test condition is true. The `Gv_waituntil()` function waits while the test condition is false. These functions do not return a value. For example:

```
Gv_alloc("count", "int");
Gv_inc("count");
Gv_waitwhile("count", "<", 20);
```

The `Gv_parallel()` function waits for a parallel variable to become greater than zero then decrements the value of the variable by one.

7.5 Using Global Variables Commands and Functions

7.5.1 Global Variables in Script Execution

Global variables are accessed by using Global Variable commands and functions with the variable name as an argument. The format for Global Variable commands is `gv_name`, and for functions, `Gv_name()`. Both must be used to make effective use of global variables.

The following example demonstrates using a Global Variable command. It would be entered at the UNIX script driver's command line:

```
$ gv_init customer int 100
```

The `gv_init` command creates a variable with the specified name, variable type, and initial value. In this example, `gv_init` is the Global Variable command, `customer` is the variable name, `int` specifies that `customer` is an integer, and `100` is the initial value of the variable `customer`.

To use a variable within a script, you should insert the `Gv_alloc()` function in the beginning of the script .c file to access the variable, for example:

```
Gv_alloc("customer", "int");
```

The `Gv_alloc()` function permits the script to access and control the variable `customer` in other global variable functions, for example:

```
Gv_protect("customer");
```

Global Variable commands may be used in a Mix command file when a multi-user load test is executed with the Mix tool. For example:

```
!gv_init users int 0
use table
start s1 s2 s3
wait
quit
```

When this Mix command file is executed, the Global Variable command `gv_init` is executed as if it was executed from the shell.

7.5.2 Global Variables Used By Multiple Scripts

To illustrate the Global Variables concept "atomic referencing," suppose one script accesses a variable to perform addition (with the `Gv_add()` function) and another script tries to access the same variable to perform multiplication (with the `Gv_mul()` function). The second script will not be able to read or operate on the variable until the first script has completed operation. This atomic referencing applies regardless of the number of scripts trying to access a variable. Each competing script will wait until it can access the variable.

However, simultaneous access to a set of variables may cause resource contention problems in your UNIX kernel when multiple scripts try to access or protect the same variable. Problems generally arise from careless use of the `Gv_protect()`, `Gv_unprotect()`, `Gv_waitwhile()`, and `Gv_waituntil()` functions.

When a script tries to access a variable protected by another script, it will wait until the variable has been unprotected before continuing with execution.

A circular protection of variables may result, causing two scripts to pause indefinitely, as in the following situations:

- script one protects variable one
- script two protects variable two
- script one tries to protect variable two
- script two tries to protect variable one

You always must be aware of the effects that variable protection will have on other running scripts. When a variable is to be used by multiple scripts but is protected by one script, you should design your scripts so that the variable becomes unprotected at some point. Other scripts attempting to read or write to the variable will block until the variable is unprotected.

When several scripts employ the `Gv_waitwhile()` or `Gv_waituntil()` function for the same test condition, some scripts may pause indefinitely. This situation may occur because the test condition becomes true (or false) for such a short time that some scripts may never see the condition for which they are waiting.

For example, a script may include the following:

```
/* Increment the global variable "users" by one */
Gv_inc("users");
/* Wait while "users" is less than 3 */
Gv_waitwhile("users", "<", 3);
/* Decrement the global variable "users" by one */
Gv_dec("users");
```

When the value of `users` becomes three, this script will return from the `Gv_waitwhile()` function. The value of `users` then will be decreased by one so that its new value is two. If other scripts are using the same `Gv_waitwhile()` function, they may not see the value of `users` reach three since it is immediately decreased by this script. Therefore, the other scripts may be blocked indefinitely. This error is avoided most easily by designing test conditions that are not true (for `Gv_waituntil()`) or false (for `Gv_waitwhile()`) for only a short time.

7.5.3 Setting Type Rate and Think Time With Global Variables

Global Variable commands and functions can be used to set emulated type rate and think time range from the UNIX script driver prior to running a test. Global variables are created to specify type rate and minimum and maximum think times. Functions in the scripts then use these variables to set type rate and think time range.

For example, to set the type rate at two characters per second and the minimum and maximum values of the think time distribution at one and five seconds, use the following Global Variable commands:

```
$ gv_init  typerate  int  2
$ gv_init  thinkmin  int  1
$ gv_init  thinkmax  int  5
```

Then, place the following statements in the beginning of the script .c file:

```
Gv_alloc("typerate", "int");
Gv_alloc("thinkmin", "int");
Gv_alloc("thinkmax", "int");

Typerate(Gv_read("typerate"));
Thinkuniform(Gv_read("thinkmin"),Gv_read("thinkmax"));
```

Each time the script is executed, a new type rate and think time distribution may be specified. If the `Typerate()` and `Thinkuniform()` functions are placed within a loop in the script, new values may be set as the script executes.

7.5.4 Examples

The following sections present introductory examples for using global variables. For simplicity, the example scripts shown in the following sections are EMPOWER scripts. Global variables also may be used in EMPOWER/X and EMPOWER/CS scripts.

7.5.4.1 Externally Initialized vs. Internally Initialized Variables

Global variables generally are created at the UNIX script driver (i.e., externally initialized) with the `gv_init` command, then accessed in a script with the `Gv_alloc()` function:

```
Gv_alloc("users", "int");
Gv_inc("users");
Gv_waitwhile("users", "<", 3);
```

To run this script three times, you could enter the following commands at the UNIX script driver:

```
$ gv_init  users  int  0
$ script  rlogin:sut  s1  &
$ script  rlogin:sut  s2  &
$ script  rlogin:sut  s3  &
```

The script may be set up so that the variable `users` is initialized inside the script:

```
Gv_alloc("users", "int");
Gv_protect("users");
if (Gv_test("users", "==", 3))
     Gv_write("users", 0);
Gv_unprotect("users");
Gv_inc("users");
Gv_waitwhile("users", "<", 3);
```

This script may then be run three times with the following transactions at the UNIX driver machine (assuming the value of `users` remains 3 between script runs). The `gv_init` command is not required at the start of the test:

```
$ script  rlogin:sut  s1  &
$ script  rlogin:sut  s2  &
$ script  rlogin:sut  s3  &
```

7.5.4.2 Synchronizing With gv_protect

In this example, three scripts are started and synchronized by accessing a variable protected at the UNIX driver machine with the `gv_protect` command. The scripts, called `s1`, `s2`, and `s3`, include the following statements:

```
Gv_alloc("synch", "int");
printf("%s ready\n", argv[0]);
Gv_inc("synch");
Xmit(".....");
    ...
Rcv("...");
```

The following example demonstrates the interaction at the UNIX script driver that will execute the scripts:

```
$ gv_init  synch  int  0
$ gv_protect  synch
$ gv_stat
EMPOWER/GV V3.1.8, Serial#R00000-000, Copyright PERFORMIX, Inc. 1988-95
 Name                Type                Value       Allocated Protector
 ------------------ ----------------- ----------- --------- ----------
 synch                    int                           0            0
CONSOLE
$ s1  &
[1] 3058
s1 ready
$ s2  &
[2] 3060
$ s3  &
[3] 3062
s2 ready
s3 ready
$ gv_stat
EMPOWER/GV V3.1.8, Serial#R00000-000, Copyright PERFORMIX, Inc. 1988-95
 Name                Type                Value       Allocated Protector
 ------------------ ----------------- ----------- --------- ----------
 synch                    int                           0            3
CONSOLE
$ gv_unprotect  synch
$ gv_stat
EMPOWER/GV V3.1.8, Serial#R00000-000, Copyright PERFORMIX, Inc. 1988-95
 Name                Type                Value       Allocated Protector
 ------------------ ----------------- ----------- --------- ----------
 synch                    int                          3          3 NONE
```

7.5.4.3 Unique Numbers

In this example, a global variable provides unique numbers for a set of scripts that delete records from a database. Since the global variable maintains its value after the scripts have executed, the scripts may be run multiple times without resetting the database.

This example uses the `gv_init` command and the `Gv_alloc()` and `Gv_inc()` functions. The `gv_init` command creates the desired global variable and sets the initial value. The `Gv_alloc()` function permits the script to use the variable and ensures that the variable type specified in the script matches the type specified in the `gv_init` command. The `Gv_inc()` function reads the current value of the variable and increments it by one.

First, the global variable `customer` is created at the command line on the UNIX script driver. The variable is an integer and the initial value is 100:

```
$ gv_init  customer  int  100
```

The following script source code deletes five records from the database:

```
/* dbdelete.c */
Empower(argc, argv)
int argc;
char *argv[];
{
int i;
int thiscustomer;
char buf[10];
Thinkuniform(2,5);      /* think 2 to 5 seconds */

/* Allocate the global variable "customer" to this script
   (if "customer" does not exist yet, exit) */
Gv_alloc("customer", "int");
Beginscenario("dbdelete");
Rcv("$ ");
Xmit("dbstartup^M");
Rcv("> ");
```

(continued on following page . . .)

```
/* delete five records from the database */
for (i = 0; i < 5; i++) {
      /* get a customer number to use and increment
         for the next script */
      thiscustomer = Gv_inc("customer");
      sprintf(buf, "%d", thiscustomer);

      Mxmit("delete ", buf, "^M", "");
      Rcv("> ");
}
Xmit("quit^M");
Rcv("$ ");
Xmit("exit^M");
Rcv("connection closed.^J");
Endscenario("dbdelete");
}
```

Two scripts containing this source code are executed. The first script is run with the display option (-d) which is shown below:

```
$ dbdelete -d rlogin:sut s1 &
$ dbdelete rlogin:sut s2 &
$

Welcome to SUT

$ dbstartup
Speedy Database started at 12:56:01 EST 10/07/90
> delete 100
record for customer 100 deleted
> delete 102
record for customer 102 deleted
> delete 103
record for customer 103 deleted
> delete 105
record for customer 105 deleted
> delete 107
record for customer 107 deleted
> quit
$ exit
connection closed.
$
```

The first deleted customer record was 100. The script used the initial value of the variable `customer` and incremented it. The second script then gained access to the variable and deleted record 101. When the first script accessed `customer` again, the value was 102. Also note that the first script was able to delete two more records (102 and 103) before the second script accessed the variable again. This is possible if the second script is delayed, for example, by a think time delay.

These scripts may be run again without resetting the database. The variable `customer` remains in shared memory with its last value, so the next run of the scripts will begin with the most current value of `customer`.

To check the current value of a global variable, use the `gv_read` command at your UNIX script driver's shell prompt:

```
$ gv_read  customer
110
$
```

If the database is restored prior to running the scripts again and you want to delete the first ten records again, `customer` must be re-initialized before running the scripts. You do not need to specify `customer` as an integer since the variable exists (the type (`int`) is optional):

```
$ gv_init  customer  int  100
```

or

```
$ gv_init  customer  100
```

7.5.4.4 Exclusive Access

In this example, several scripts run word processing operations to edit a file called "`proposal`". While one script edits the file, a global variable protects the file so that other scripts can not access it. When the script completes editing, the file is unprotected.

A global variable is protected with the function `Gv_protect()` and unprotected with the function `Gv_unprotect()`. In our example, a variable called `filetoken` is created. When a script starts to edit the file, `filetoken` is protected. If another script has already protected the variable (and is editing the file), the `Gv_protect()` function will be unsuccessful. The script will wait until it can protect the variable and begin editing.

Before the emulation is executed, the variable must be created:

```
$ gv_init filetoken int 0
```

The value of `filetoken` is never used because we are protecting (not testing or incrementing) the variable. Therefore, its initial value is unimportant. Also, the variable does not need to be re-initialized for subsequent runs of the emulation.

Each script will include lines similar to the following example for the file editing session. Editing must be surrounded by the `Gv_protect()` and `Gv_unprotect()` statements which are shown below:

```
/* wordproc_1.c */
/* Allocate the global variable "filetoken" to this script
   (if "filetoken" does not exist yet, exit) */
Gv_alloc("filetoken", "int");
Beginscenario("wordproc_1");
Rcv("$ ");
/* do some transactions - not detailed here */
Xmit("cd documents^M");
   .
   .
   .
Rcv("$ ");
/* Protect variable "filetoken" from other scripts
   (Only one script can protect filetoken at a time) */
Gv_protect("filetoken");
/* do the exclusive edit - not detailed here */
Xmit("vi proposal^M");
```

(continued on following page...)

```
   .
   .
Xmit(":wq^M");
Rcv("$ ");

/* Unprotect variable "filetoken"
   (This allows another script to protect "filetoken") */
Gv_unprotect("filetoken");
Xmit("exit^M");
Rcv("connection closed.^J");
Endscenario("wordproc_1");
```

7.5.4.5 Indefinite Benchmark Duration

In this example, several scripts are designed to run indefinitely so that they may be terminated from the UNIX script driver The `kill` command of Mix does not ensure that scripts will terminate gracefully because the script simply disconnects from the system under test (SUT). You can assign a global variable that instructs scripts to complete the current series of transactions and then quit the SUT application before disconnecting.

To set up this type of script control, each script should be edited to check the designated global variable before each series of transactions begins. This verification is accomplished by enclosing each series of transactions in a `while` loop and using the `Gv_test()` function which tests the value of the global variable. If the comparison is true, then the script will proceed with the transactions.

First, the global variable is created:

```
$ gv_init workflag int 1
```

The variable is given an initial value of 1. The name `workflag` was chosen because the variable specifies whether or not to continue working.

The `Gv_test()` function makes a relational comparison between the value of the variable and a specified value. `Gv_test()` returns the value 1 if the comparison is true, and 0 if it is false.

The following interaction produces these results, using the `gv_test` command at the shell prompt:

```
$ gv_init a int 5
$ gv_test a "<" 1
0
$ gv_test a ">" 1
1
```

The script code using `Gv_test()` is shown below:

```
Gv_alloc("workflag", "int");
Beginscenario("worker");
Rcv("$ ");
/* enter the SUT application */
Xmit("officemail^M");
Rcv("om> ");
while ( Gv_test("workflag", "==", 1) ) {

     /* enter a series of transactions - not detailed here */
     Xmit("....");
            .
            .
     Rcv("om> ");
}

/* quit the SUT application */
Xmit("quit^M");
Rcv("$ ");
/* exit from the SUT */
Xmit("exit^M");
Rcv("connection closed.^J");
Endscenario("worker");
```

Each script in the multi-user test should have some form of this code. Contents of the `while` loop would differ for each series of transactions. When you are ready to terminate all scripts, redefine the value of the variable `workflag` to be 0 with the `gv_write` command entered at the UNIX script driver:

```
$ gv_write  workflag  0
1
```

This command will make each script wait until the current series of transactions is complete, then quit the SUT application. The system responds to the `gv_write` command with the original value of the variable, 1.

Before running the scripts again, `workflag` will have to be reset. You are not required to specify `workflag` as an integer because it already exists:

```
$ gv_init  workflag  1
```

If the value of `workflag` is not reset, each script would fail the test specified in `Gv_test()` and quit without processing any transactions.

Use the `gv_init` command to create a new variable and assign a value to the variable. Unless the specified variable currently is alloacted by a script, `gv_init` also can assign a new value to an existing variable.

By comparison, you can use the `gv_write` command to assign a new value to an existing variable, even if the variable already is allocated by a script.

Therefore, `gv_init` generally is used between tests, and `gv_write` is used during a test.

7.5.4.6 Fixed Number of Transactions

In the following example, a set of scripts will execute a series of transactions a specified number of times. As in the previous example, the scripts are set up so that the value of a global variable is checked before the transactions execute.

A global variable function is created to count down how many times the series of transactions executes. The function `Gv_dec()` is used to decrement the count each time the transactions execute.

First, the variable is created and given the initial value 600, which is the total number of times the series of transactions will be executed during the multi-user emulation.

```
$ gv_init work int 600
```

Each script first gains access to the variable `work` with the `Gv_alloc()` function. Then, the `Gv_dec()` function decrements the value of `work` by one. To determine if the new value of `work` is greater than zero, a test is performed with the `while` statement. If the value of the variable is greater than zero, the contents of the loop (the series of transactions) are executed.

Each of these scripts will contain the following:

```
Gv_alloc("work", "int");
Beginscenario("worker");
Rcv("$ ");
/* enter the SUT application */
Xmit("officemail^M");
Rcv("om> ");
while (Gv_dec("work") > 0) { /* there's work to do */
      /* enter a series of transactions - not detailed here */
      Xmit("....");
              .
              .
      Rcv("om> ");
}
/* quit the SUT application */
Xmit("quit^M");
Rcv("$ ");
/* exit from the SUT */
Xmit("exit^M");
Rcv("connection closed.^J");
Endscenario("worker");
```

Once the value of `work` becomes zero (i.e., the series of transactions has been executed 600 times), each script will terminate as it gets to the `while` loop. As each script falls through the `while` loop, the value of `work` is decremented to an increasingly large *negative* number. Since the `while` loop is set up to execute only if `work` is greater than zero, this negative number creates no problems.

To run the multi-user test again, the variable `work` must be reset. If desired, you may initialize `work` to a different value, and you are not required to specify `work` as an integer since it already exists:

```
$ gv_init work 120
```

7.5.4.7 Script Synchronization

In this example, a set of scripts is synchronized before the scripts execute commands on the SUT. A global variable is created and incremented as each script starts. The function `Gv_waitwhile()` pauses each script until all scripts have started.

The `Gv_waitwhile()` function makes the script pause as long as the specified relational comparison is true. When the comparison becomes false, script execution continues past the `Gv_waitwhile()` statement.

The global variable that stores the number of executed scripts is initialized:

```
$ gv_init users int 0
```

Each script first gains access to the variable `users` with the `Gv_alloc()` function. Then, the `Gv_inc()` function increments the value of `users` by one and `Gv_waitwhile()` tests the value of `users`.

Each script includes the following:

```
/* Allocate the global variable "users" to this script
   (if "users" does not exist yet, exit) */
Gv_alloc("users", "int");

Beginscenario("datetest");

Rcv("$ ");

/* Increment the global variable "users" */
Gv_inc("users");

/* Wait while "users" is less than 3 */
Gv_waitwhile("users", "<", 3);

/* do some transactions - not detailed here */
Xmit("cd documents^M");
   .
   .
   .
Rcv("$ ");

Xmit("exit^M");
Rcv("connection closed.^J");

Endscenario("datetest");
```

To run the multi-user emulation again, the variable `users` must be reset since it will retain a value of three after the first execution. You are not required to specify that `users` is an integer since it already exists.

Example:

```
$ gv_init users 0
```

7.5.4.8 Limited Script Activity in Multi-User Emulation

In this example, parallel Global Variables control execution of multiple scripts. Parallel variable commands and functions allow a subset of emulated users to execute a portion of the script at a given time. Specifically, during an emulation involving 500 users, a portion of the test may be executed in parallel by only 50 users. Each user will execute one script. The Global Variable `worker` is created from the UNIX script driver shell as a parallel type variable with an initial value of 50:

```
$ gv_init  worker  parallel  50
```

The following script is executed by all users (some detail is left out for brevity):

```
login()

{/* login transactions - not detailed here */}

Empower(argc, argv)
      int argc;
      char *argv[];
{
      Gv_alloc("worker", "parallel");
      login();
      Beginscenario("example");

      /* ... numerous transactions */

      Gv_parallel("worker");

      /* ... limited portion of transactions */

      Gv_unparallel("worker");

      /* ... numerous transactions */

      Endscenario("example");
}
```

The first 50 users to reach the `Gv_parallel()` function during script execution immediately will execute the limited portion of transactions. The remaining 450 users will reach the `Gv_parallel()` function and will wait.

As the 50th user executes the `Gv_parallel()` function, the value of the variable `worker` is decremented to zero. As each of the first 50 scripts reaches the `Gv_unparallel()` function, the value of `worker` is incremented to be greater than zero, so that one of the waiting scripts can execute the limited portion of transactions. This process ensures that only the first 50 scripts will execute the specified transactions at the same time.

7.5.4.9 Sharing a File Offset

Creating scripts to read input files on the UNIX script driver is a common practice. Such files contain names, addresses, phone numbers, etc., used for emulating database queries and updates. For such an emulation, each user would need unique data, so, each script must read from a different file. A test of 100 users would need 100 input files. Considerable effort often is required to create the 100 files, especially if you must run tests with different numbers of users.

The EMPOWER products provide a convenient way for many scripts to read from a single file ensuring that data for each script is unique. You can use the File I/O functions (such as `Fioreadline()`) in scripts to read and write files. (See Section 7 Script Content and Enhancement of your Script Development manual for more information on File I/O functions.) These functions include a `Fioshare()` function which is used to identify files that are to be shared. `Fioshare()` must be called before any other File I/O functions are called to reference the same file.

`Fioshare()` in a script presumes execution of the `fioshare` command at the UNIX script driver's shell prompt. The `fioshare` command creates a global variable that contains the offset for the next byte to be read from a shared file. The value of the global variable (offset) remains between tests, so you can continue to read an input file from the point left by the previous test. Saving the offset in this manner is useful in tests that corrupt a database on the SUT. The ability to avoid the same transactions means you do not have to restore the database before every test.

You must execute the `fioshare` command if you want to resume reading from the beginning of the input file. For this reason, `fioshare` often is run from Mix command files that set up for a new test.

For example, assume we require scripts to read commands from the following input file `cmds`:

```
date
who
ls /bin
ps -ef
grep xxx /etc/passwd
```

The following command will create the global offset for the file:

```
$ fioshare  cmds
```

A segment in the script described above might look like the following example. Each execution of this script will read different lines.

```
Fioshare("cmds");
Fioreadline("cmds");
while (FIOLEN != -1) {
    Mxmit(FIOBUFFER, "^M", "");
    Rcv("$ ");
    Fioreadline("cmds");
}
```

The global offset is stored in a global variable and the global variable's name is the `inode` of the shared file. This condition can be confirmed by typing the "`ls -i`" command with an argument of the shared file and then running the `gv_stat` command.

For example:

```
$ fioshare  cmds
$ ls  -i  cmds
59449 cmds
$ gv_stat
EMPOWER/GV V3.1.8, Serial#R00000-000, Copyright PERFORMIX, Inc. 1988-95
 Name                 Type                 Value       Allocated Protector
 -------------------- ------------------- ----------- --------- ----------
 59449                long int                       0         0
$
```

To discontinue sharing a file, the `Fiounshare()` function and `fiounshare` shell command are used. Neither the function nor the command remove the global variable offset for the file. They simply mark the global variable as being unusable for further File I/O functions.

The `Fiounshare()` function disables the sharing of the file offset only for the script that executes the function. The `fiounshare` shell command disables sharing of the file offset for all scripts currently sharing the file.

To remove the global variable offset, you must use the `gv_rm` shell command to remove the global variable named in the `gv_stat` output listed above. For example:

```
$ gv_rm  -f  59449
```

7.5.4.10 Script Scenario Selection

For this example, we use a global variable to select types of scenarios to execute. Multiple copies of a single script are used and the script contains all of the desired scenarios.

The test includes a mix of following scenarios:

Scenario	Count
receptionist	1 user
database	3 users
manager	1 user
visitor	Remainder of users

A single script with functions containing each set of transactions will implement this mix of scenarios. By using a global variable called `users`, the `EMPOWER()` function will select a unique ID for the script. Global Variables allow this test to execute without passing arguments to the script through the Mix table.

The `Gv_inc()` function increments the value of `users`. Before it is incremented, the value of `users` is saved in the local variable `myid`. Each script has a unique value of `myid` which is used in a `switch()` statement to select the proper scenario to run.

First, the global variable `users` is initialized with a zero value. *Note:* This initialization must be performed prior to each run:

```
$ gv_init users int 0
```

Several copies of the script are started with Mix. The first script to call `Gv_inc()` returns a value of 0 to `myid`, so it calls the `receptionist()` function. The next three (`myid` values 1, 2, and 3) call the `database()` function. The next script (`myid` = 4) calls the `manager()` function. The remainder of scripts (`myid` = 5, 6, etc.) call the `visitor()` function.

In the following script, the actual transactions for the four scenarios are left out for brevity:

```
login()
{
Rcv("login: ");

Xmit("userid^M");
Rcv("word:");

Xmit("passwd^M");
Rcv("$ ");
}

manager()
{  /* manager transactions go here */  }

receptionist()
{  /* receptionist transactions go here */  }

database()
{  /* database transactions go here */  }

visitor()
{  /* visitor transactions go here */  }

Empower()
{
     int myid;

     Gv_alloc("users", "int");
     myid = Gv_inc("users");

     Timeout(30, CONTINUE);
     Thinkuniform(1,3);

     login();

     switch(myid) {
          case 0: Beginscenario("receptionist");
               receptionist();
               Endscenario("receptionist");
               break;
```

(continued on following page . . .)

```
            case 1:
            case 2:
            case 3: Beginscenario("database");
                 database();
                 Endscenario("database");
                 break;
            case 4: Beginscenario("manager");
                 manager();
                 Endscenario("manager");
                 break;
            default: Beginscenario("visitor");
                 visitor();
                 Endscenario("visitor");
                 break;
       }
       Xmit("exit^M");
       Rcv("");
}
```

7.5.4.11 Scripts Controlled By One Script

In the following example, 11 identical scripts execute simultaneously and are controlled by the first executed script. This controlling script will not interact with SUT applications so its communication port to the SUT can be specified as "pseudo:sh -i" on the command line. The controlling script sets the global variables that are accessed by the other scripts.

First, four global variables are initialized:

```
$ gv_init control int 0
$ gv_init quitflag int 0
$ gv_init users int 0
$ gv_init duration int 3600
```

The first script to call Gv_inc() becomes the controller. It pauses (with Gv_waitwhile()) while the other ten scripts start. As each script starts, this controlling script increments the variable users and waits while users is less than ten. Once these ten scripts have started, the controller script uses `Gv_read()` to retrieve the value of global variable `duration`. The controller script then executes a `Sleep` delay for the number of seconds specified by the `duration` variable.

All other scripts execute the function `do_invoice`. This function includes a `while` loop which checks the value of global variable `quitflag`. If `quitflag` has the value zero, the function continues to execute.

When the controller script concludes the `Sleep` delay, it increments `quitflag`, causing the other scripts to return from `do_invoice`. The controller script uses `Gv_waitwhile()` to pause until the other scripts have completed. Each of the other scripts decrements the variable `users`. When the value of `users` is zero, the controller script returns from `Gv_waitwhile()`, re-initializes `control` and `quitflag`, and exits.

Each script would include the following:

```
login()
{/* login transactions - not detailed here */}

do_invoice()
{
     /* do some transactions - not detailed here */
     Xmit("...");
         .
         .
     Rcv("$ ");
}

Empower()
{
     Gv_alloc("control", "int");
     Gv_alloc("quitflag", "int");
     Gv_alloc("users", "int");
```

(continued on following page . . .)

```
     if (Gv_inc("control") == 0) {
     Gv_waitwhile("users", "<", 10);
     Gv_alloc("duration", "int");
     Sleep(Gv_read("duration"));
     Gv_inc("quitflag");
     Gv_waituntil("users", "==", 0);
     Gv_write("control", 0);
     Gv_write("users", 0);
     Gv_write("quitflag", 0);
     }
     else {
     login();

     Beginscenario("invoice");

     Gv_inc("users");

     while(Gv_test("quitflag", "==", 0))/* check for quit indicator */
          do_invoice();

     Gv_dec("users");

     Endscenario("invoice");

     Xmit("exit^M");
     Rcv("connection closed.^J");
    }
}
```

7.5.4.12 Scripts With Collective Throughput

In the following example, scripts use global variables to control throughput of transactions the scripts submit to the SUT. Script execution is controlled at the shell by setting the value of the global variable `workflag`.

First, global variables `workflag` and `transactions` are initialized:

```
$ gv_init  workflag  int  0
$ gv_init  transactions  int  0
```

As each script executes, it waits while `workflag` has a zero value which allows scripts to synchronize before continuing. Continuation is prompted at the shell by setting the value of `workflag` to one with `gv_write`.

When each script returns from `Gv_waitwhile()`, it uses the `Time()` function to record system time in the local variable `starttime`. Then, the script checks `workflag` with `Gv_read()` and if the value of `workflag` is one, enters a `while` loop to begin a transaction. The transaction executes, the variable `transactions` is incremented, and the time is stored in the variable `curtime` with `Time()`.

`Gv_read()` retrieves the number of transactions in the variable `transactions`. This number is divided by the difference between `starttime` and `curtime` to determine the cumulative throughput for all scripts. If the cumulative throughput exceeds the target specified by `THRUPUT`, the script will execute a `Sleep` delay for zero to three seconds.

Script execution terminates when the shell command `gv_write` resets the value of `workflag` to zero.

An example script follows:

```
#define THRUPUT 10.0   /* transactions per second */
login()
{/* login transactions - not detailed here */}

Empower()
{
     struct timevalue starttime, curtime;
     Gv_alloc("transactions", "int");
     Gv_alloc("workflag", "int");

     Timeout(30, CONTINUE);
     Thinkuniform(1,3);
     login();

     Gv_waitwhile("workflag", "==", 0);
     Beginscenario("worker");

     Time(&starttime);
     while(Gv_test("workflag", "==", 1)) {
          /* do a transaction - not detailed here */
          Xmit("...");
          Rcv("$ ");

          Gv_inc("transactions");

          Time(&curtime);
          if ((Gv_read("transactions")  /
             Difftime(&starttime, &curtime)) > THRUPUT)
             Sleep(Range(0,3));
     }
     Endscenario("worker");

Xmit("exit^M");
Rcv("");
}
```

0095794.102600

7.5.4.13 Passing SUT Data to the Next Script

In the following example, two scripts process transactions with a database application. After the first script has completed database operations, a control number received from the SUT is saved in a global variable for the second script's use. The first script copies the control number from the response buffer RBUFFER into the global variable mailbox. The second script copies the saved control number from the mailbox into a transmit buffer.

First, the global variable is created at the UNIX script driver:

```
$ gv_init mailbox str ""
```

The first script includes the following entries. Notice that the control number from the SUT is saved in the global variable controlnum.

```
char control[GVSTRINGMAX + 1];
Gv_alloc("mailbox", "str");

Rcv("$ ");

/* do a transaction - not detailed here */
Xmit("...");
/* align the control number at the beginning of RBUFFER */
Scan("Control number is: ");

/* read control number response into RBUFFER */
Rcv("$ ");

strncpy(control, RBUFFER, 10);

/* put control number in global variable for script2 */
Gv_writev("mailbox", control, (char*)NULL);
```

The second script uses the control number string from the first script:

```
Gv_alloc("mailbox", "str");
char control[GVSTRINGMAX + 1];

Rcv("$ ");

/* read the control number string left in the global variable */
Gv_readv("mailbox", control);

/* do a transaction with the new string */
Mxmit(control, "^M", "");
Rcv("$ ");
```

[This page intentionally left blank]